**Tutorial on artificial neural network**

Loc Nguyen

Loc Nguyen’s Academic Network, Vietnam

Email: ng\_phloc@yahoo.com

Homepage: www.locnguyen.net

# Abstract

It is undoubtful that artificial intelligence (AI) is being the trend of computer science and this trend is still ongoing in the far future even though technologies are being developed suddenly fast because computer science does not reach the limitation of approaching biological world yet. Machine learning (ML), which is a branch of AI, is a spearhead but not a key of AI because it sets first bricks to build up an infinitely long bridge from computer to human intelligence, but it is also vulnerable to environmental changes or input errors. There are three typical types of ML such as supervised learning, unsupervised learning, and reinforcement learning (RL) where RL, which is adapt progressively to environmental changes, can alleviate vulnerability of machine learning but only RL is not enough because the resilience of RL is based on iterative adjustment technique, not based on naturally inherent aspects like data mining approaches and moreover, mathematical fundamentals of RL lean forwards swing of stochastic process. Fortunately, artificial neural network, or neural network (NN) in short, can support all three types of ML including supervised learning, unsupervised learning, and RL where the implicitly regressive mechanism with high order through many layers under NN can improve the resilience of ML. Moreover, applications of NN are plentiful and multiform because three ML types are supported by NN; besides, NN training by backpropagation algorithm is simple and effective, especially for sample of data stream. Therefore, this study research is an introduction to NN with easily understandable explanations about mathematical aspects under NN as a beginning of stepping into deep learning which is based on multilayer NN. Deep learning, which is producing amazing results in the world of AI, is undoubtfully being both spearhead and key of ML with expectation that ML improved itself by deep learning will become both spearhead and key of AI, but this expectation is only for ML researchers because there are many AI subdomains are being invented and developed in such a way that we cannot understand exhaustedly. It is more important to recall that NN, which essentially simulates human neuron system, is appropriate to the philosophy of ML that constructs an infinitely long bridge from computer to human intelligence.

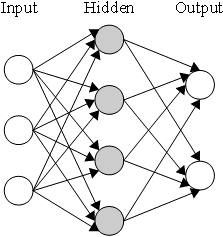
**Keywords:** artificial neural network (ANN), neural network (NN), machine learning (ML), artificial intelligence (AI).

# 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 parameter 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.

Please distinguish input unit from input and distinguish output unit from output because input is the input value of any unit and output is the output value of any unit. These are conventions in this research. 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*.



**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 report research 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 as usual. 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 composition of other hidden units which in turn are 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 amplitude of output of a unit. This activity aims to determine and assert output of a unit. Note that outputs of previous units are inputs of current unit.

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

Diagram

Description automatically generated

**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 output of a unit, we have following equation (Han & Kamber, 2006, p. 331), (Kröse & Smagt, 1996, pp. 16-17) for computing output value of a unit.

|  |  |
| --- | --- |
|  | (1.1) |

Or shortly:

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

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

|  |  |
| --- | --- |
|  | (1.2) |

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 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*, propagation rule at time point *t* + 1 is rewritten as follows:

|  |  |
| --- | --- |
|  | (1.3) |

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 *k* 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. 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 units *j* 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 inputs modified by their respective weights. Summing function or adder is responsible for this summing task.
* Its output *yk* is 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*, propagation rule is rewritten exactly as follows:

|  |  |
| --- | --- |
|  | (1.4) |

Where *N*(*k*) denotes a set of previous (clamped) units to which the current clamped unit *k* connects. Given time point *t*, propagation rule is rewritten fully as follows:

Propagation rule essentially transforms inputs to outputs but an output *yk* may not totally equal to 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 other 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 composes 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.

Activation function *f*(*.*), which is an important factor of NN, is 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]. A popular formula of sigmoid function is:

|  |  |
| --- | --- |
|  | (1.5) |

Where *e*(.) or exp(.) denotes exponent function. Exponential 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:

|  |  |
| --- | --- |
|  | (1.6) |

The equation above is called *weight update rule* and hence, weight update rule focuses on how to calculate 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:

|  |  |
| --- | --- |
|  | (1.7) |

The equation above is called *bias update rule* and hence, bias update rule focuses on how to calculate 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 weight update rule and bias update rule are based on weight deviation and bias deviation, these deviations 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):

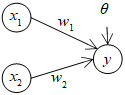
|  |  |
| --- | --- |
|  | (1.8) |

Where the positive constant *γ* which is called learning rate (0 < *γ* ≤ 1) specifies power of the proportionality, 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 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):

|  |  |
| --- | --- |
|  | (1.9) |

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 as 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):

|  |  |
| --- | --- |
|  | (1.10) |

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

|  |  |
| --- | --- |
|  | (1.11) |

Therefore, weight update rule of Perceptron is slightly similar to Hebbian rule. Bias update rule of Perceptron is:

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

|  |  |
| --- | --- |
|  | (1.12) |

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 the input units *xi* (s). Propagation rule of Adaline is (Kröse & Smagt, 1996, p. 28):

|  |  |
| --- | --- |
|  | (1.13) |

Obviously, 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):

|  |  |
| --- | --- |
|  | (1.14) |

Where (Kröse & Smagt, 1996, p. 28),

|  |  |
| --- | --- |
|  | (1.15) |

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 least squares method, the optimal (*wi*\*\*, *θ*\*\*)*T* is 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*).

|  |  |
| --- | --- |
|  | (1.16) |

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*).

|  |  |
| --- | --- |
|  | (1.17) |

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 *γ* and the gradient of *ε*(*p*)(*wi*, *θ*) as follows (Kröse & Smagt, 1996, p. 29):

|  |  |
| --- | --- |
|  | (1.18) |

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:

|  |  |
| --- | --- |
|  | (1.19) |

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):

|  |  |
| --- | --- |
|  | (1.20) |

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. The squared error function is also called loss function. 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 Lipschitz continuous second-order functions which are fed to SGD, which will be explained in the next section mentioning convergence of SGD in detail.

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

Recall that weight deviation Δ*who* and 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 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 weight deviation Δ*who* and bias deviation Δ*θo* of any output unit as follows:

|  |  |
| --- | --- |
|  | (1.21) |

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

Let (Kröse & Smagt, 1996, p. 34),

|  |  |
| --- | --- |
|  | (1.22) |

The quantity *δo* is called error of output unit in literature. The derivative *f*’(*xo*) should be replaced by *f*’(*yo*) because *xo* should be squashed into *yo* so that it will not be out of value space. As a result, the quantity *δo* is improved as follows:

We have the succinct equation of weight deviation Δ*who* and bias deviation Δ*θo*.

|  |  |
| --- | --- |
|  | (1.23) |

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 hidden unit *h* is connected to a set of output units *o*. Therefore, the squared error function *ε*(*yh*) of hidden unit *h* is the sum of output errors *ε*(*yo*) with regard to such set of output units, as follows:

|  |  |
| --- | --- |
|  | (1.24) |

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 Lipschitz continuous second-order functions which are fed to SGD.

Where,

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

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

|  |  |
| --- | --- |
|  | (1.25) |

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:

|  |  |
| --- | --- |
|  | (1.26) |

Where,

|  |  |
| --- | --- |
|  | (1.27) |

Note,

The quantity *δh* is called error of hidden unit in literature. The equation above is an extension of delta rule. The derivatives *f*’(*xo*) and *f*’(*xh*) should be replaced by *f*’(*yo*) and *f*’(*yh*) because *xo* and *xh* should be squashed into *yo* and *yh* so that they will not be out of value space. As a result, the quantities *δo* and *δh* are improved as follows:

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:

The derivative *f*’(*xk*) should be replaced by *f*’(*yk*) because *xk* should be squashed into *yk* so that it will not be out of value space. As a result, the derivative *f*’(*xk*) is improved as follows:

Where,

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

Where,

|  |  |
| --- | --- |
|  | (1.28) |

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 output of unit *i*, input of unit *j*, and output of unit *j*. Obviously, we have *Oi* = *yi*, *Ij* = *xj* = *sj*, and *Oj* = *yj*. These notations are necessary for describing pseudo code of backpropagation algorithm because output units and hidden units in some cases 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. Propagation rule is written according to these notations (Han & Kamber, 2006, p. 331) for computing the output value of a unit as follows:

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:

Backpropagation algorithm (backward propagation algorithm) is described here along with an example of document classification (Nguyen, 2022), which is implementation of propagation rule, weight update rule, and bias update rule. Suppose a sample consists of many data rows and each row has many attributes. There is a so-called class attribute which is used to group (classify) rows. All attributes except the class attribute are often represented as input units in NN and the class attribute is often represented as output unit in NN. When feedforward NN is used to classify document then, rows represent documents and non-class attributes are terms; in this case, the sample becomes a matrix *n*x*p*, which have *n* rows and *p* columns with respect to *n* document vectors and *p* terms. This sample for document classification is called *corpus*. Backpropagation algorithm (Han & Kamber, 2006, pp. 330-333) is also a famous supervised learning algorithm for classification, besides learning feedforward NN. Therefore, backpropagation algorithm here is applied to classify the corpus as an example of supervised learning by NN (Nguyen, 2022). It processes iteratively data rows in training corpus and compares network’s prediction for each row to actual class of the row. For each time it feeds a training row, weights are modified in order to minimize 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 backpropagation algorithm for learning NN by pseudo-code like programming language.

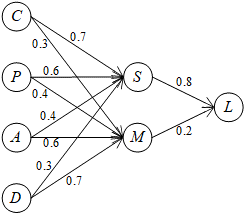
|  |
| --- |
| **1. Initializing the weights**: Weights *wij* of all connections between units are initialized as random real numbers which should be in space [0, 1]. Each bias *θi* associated to each unit is also initialized, which is 0 as usual.  *While terminating condition is not satisfied*  *For 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 output value of unit *i* from previous layer and *θj* is 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 all 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 the figure above, 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 backpropagation algorithm. It is required to compute the output values *OS*, *OM*, *OL* and update connection weights. For simplicity, activation function is sigmoid function . According to propagation rule (Han & Kamber, 2006, p. 331) for computing 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 value of output unit *L*. Because *D*1 belongs to “*computer science*” class, we have:

Let *ErrL*, *ErrS*, and *ErrM* be 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:

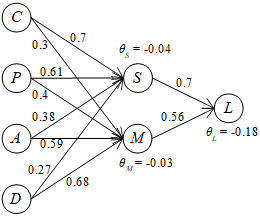
According to the equation for updating biases *θS*, *θM*, and *θL*, we have:

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 the output values *OS*, *OM*, and *OL*. For simplicity, activation 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. More about learning algorithm

Given an artificial neural network (ANN), the set of inputs *x*1, *x*2,…, *xk*,… is denoted as ***x*** = (*x*1, *x*2,…, *xk*,…)*T* which is called *input vector* and the set of outputs *y*1, *y*2,…, *yk*,… is denoted as ***y*** = (*y*1, *y*2,…, *yk*,…)*T* which is called *output vector*, whereas *propagation rule* which is the core of ANN process is specified as follows:

Anyhow propagation rule represents ANN itself. Indeed, propagation rule analyzes ANN process into scalar values for simplicity and thus, propagation rule is generalized by matrix operators as follows:

|  |  |
| --- | --- |
|  | (2.1) |

Note, *f*(.) is vector-by-vector function whose each its element is *activation function* for scalar variable as usual, which should be denoted as ***f***(.) but the unusual notation *f*(.) is kept intact for simplicity. Recall that there are two rules for learning ANN such as Hebbian rule and delta rule, in which Hebbian rule develops into delta rule which in turn develops into stochastic gradient descent (SGD) algorithm. Moreover, SGD is associated with backpropagation algorithm to make learning ANN more effective and feasible in deep learning. Conversely, delta rule is derived from SGD. This section focuses on association of SGD and backpropagation algorithm in most general techniques with vector and matrix, where learning ANN is often called training ANN.

Thevector variables ***x***, ***y***, and ***z*** represent general vector layers so that ANN is represented 3-layer ANN whose parameters are *U* and *V* are matrices whereas Θ, and Φ are vectors. Note, the superscript “*T*” denotes transposition operator of vector and matrix.

Of course, dimensions of ***x***, *U*, Θ, , ***y***, *V*, Φ, , and **z** are *n*x1, *m*x*n*, *m*x1, *m*x1, *m*x1, *q*x*m*, *q*x1, *q*x1, and *q*x1, respectively. Activation functions *f*(), *f*() are easily extended within context of vector operation when activation function acts on every elements of and , which means that the essence of activation function *f*(.) is kept intact. Therefore, *f*() and *f*() are also called activation vectors.

|  |  |
| --- | --- |
|  | (2.2) |

|  |  |
| --- | --- |
|  | (2.3) |

Where indices “***y***” and “***z***” in *f****y***() and *f****z***() implies the layers “***y***” and “***z***”, respective. Note, *f*() and *f*() are scalar activation function as usual, like *sigmoid function*, for instance:

Note, *f*() and *f*() are *m*x1 vector and *q*x1 vector, respectively. Derivatives of activation vectors *f*() and *f*() are determined by taking derivative of activation function on transposition of ***x***’ and ***y***’, which are *Jacobian matrices* *f*’(), *f*’().

|  |  |
| --- | --- |
|  | (2.4) |

|  |  |
| --- | --- |
|  | (2.5) |

Where,

Dimensions of Jacobian matrices *f*’() and *f*’() are *m*xm and *q*x*q*, respectively. Note, *f*’() and *f*’() are first-order derivatives of scalar activation function as usual, like derivative of sigmoid function, for instance:

*Feedforward network* (*FNN*) with vector layers represented by a series of layers ***x***0, ***x***1, ***x***2,…, ***x****K* is called *K*-layer FNN which is represented as follows:

By extending notation of ANN with vector operators, how to learn ANN by *stochastic gradient descent* (*SGD*) algorithm associated *backpropagation algorithm* is not so difficult to be extended for estimating weight matrices *U*, *V* and bias matrices Θ, Φ.

Within backpropagation algorithm, *V* and Φ are estimated before *U* and Θ because vectors ***x***, ***y***, and ***z*** are input layer, hidden layer, and output layer, respectively without loss of generality. In general case, *likelihood* of ANN denoted *l*(***z*** | *U*, Θ, *V*, Φ) is defined so as to estimate *U*, Θ, *V*, and Φ by maximizing *L*(***z*** | *U*, Θ, *V*, Φ) with SGD when *l*(***z*** | *U*, Θ, *V*, Φ) is function of *U*, Θ, *V*, and Φ given data ***z***. Let the dot “*.*” denote the set of parameters {*U*, Θ, *V*, Φ} and so, let *l*(***z*** | *U*, Θ, *V*, Φ) = *l*(***z*** | *.*) = *l*(***z***)

Which implies within context of SGD:

Note, ∇*Ul*(***z***), ∇Φ*l*(***z***), ∇*Vl*(***z***), and ∇Φ*l*(***z***) are *gradients* (first-order derivatives) of likelihood *l*(***z***) with regard to *U*, Θ, *V*, and Φ, respectively whereas *γ* (0 < *γ* ≤ 1) is learning rate and the superscript “*T*” denotes transposition operator of matrix and vector. Therefore, the main problem which is solved is how to calculate these gradients.

As usual, *likelihood* *l*(***Z***) is defined as the negative of squared norm given the real output ***z***’.

|  |  |
| --- | --- |
|  | (2.6) |

Where,

According to literature, differential of *l*(***z***) with regard to ***z*** is:

Where tr(.) denotes *trace operator* of square matrix. Gradient of *l*(z) with regard to ***z*** is:

Differential of *l*(***z***) with regard to *V* is:

Therefore, ∇*Vl*(***z***) is:

Differential of *l*(***z***) with regard to Φ is:

Therefore, ∇Φ*l*(***z***) is:

This implies:

Differential of *l*(***z***) with regard to ***y*** is:

Therefore, gradient of *l*(**z**) with regard to ***y*** denoted ∇***y****l*(***z***) is:

Differential of *l*(***z***) with regard to *U* is:

Therefore, ∇*Ul*(***z***) is:

Differential of *l*(***z***) with regard to Θ is:

Therefore, ∇Θ*l*(***z***) is:

This implies:

As a result, gradients ∇*Ul*(***z***), ∇Φ*l*(***z***), ∇*Vl*(***z***), and ∇Φ*l*(***z***) which are necessary for association of SGD and backpropagation algorithm are specified as follows:

|  |  |
| --- | --- |
|  | (2.7) |

Such that:

Note, ∇Φ*l*(***z***), ∇*Vl*(***z***), and ∇Φ*l*(***z***) are column vectors, which does not follow convention of numerator layout, for easy explanation.

In the general case of feedforward network (FNN) with vector layers represented by a series of layers ***x***0, ***x***1, ***x***2,…, ***x****K* is called *K*-layer FNN as follows:

|  |  |
| --- | --- |
|  | (2.8) |

It is easy to extended estimation equations of weight parameters *Wk* and bias parameter Θ*k* given likelihood *l* = *l*(***x****k*). For *k* = *K* at the last layer (output layer) given real output from environment, gradients and at the last layer *K* are estimated as follows:

|  |  |
| --- | --- |
|  | (2.9) |

Such that:

For all remaining *k* (s) such that 1 ≤ *k* ≤ *K*–1 at all hidden layers, gradients and are estimated as follows:

|  |  |
| --- | --- |
|  | (2.10) |

Such that:

Where *γ* (0 < *γ* ≤ 1) is learning rate. The subscripts ***x****k* in the derivatives implies the layer ***x****k* of function *f*(.). If *l*(.) is likelihood function, the gradient is called reward at layer ***X****k* and its negative is called error at layer ***X****k*. Moreover, please pay attention that and are column vectors here for easy computation although they are row vectors according to numerator layout convention because they are gradients of scalar-by-vector function.

In the most general case, ***x***, ***y***, and ***z*** become *input matrix* ***X***, *hidden matrix* ***Y***, and *output matrix* **Z** representing general matrix layers so that ANN is represented 3-layer ANN as follows:

|  |  |
| --- | --- |
|  | (2.11) |

Where parameters of ANN are weight matrices *U*, *V*, *P*, *Q* and bias matrices Θ, Φ.

Of course, dimensions of ***X***, *P*, *Q*, Θ, , ***Y***, *U*, *V*, Φ, , and **Z** are *n*x*r*, *m*x*n*, *r*x*s*, *m*x*s*, *m*x*s*, *m*x*s*, *p*x*m*, *s*x*q*, *p*x*q*, *p*x*q*, and *p*x*q*, respectively. Activation functions *f*(), *f*() are easily extended within context of matrix operation when activation function acts on every elements of and , which means that the essence of activation function *f*(.) is kept intact. Therefore, *f*() and *f*() are also called activation matrices.

|  |  |
| --- | --- |
|  | (2.12) |

|  |  |
| --- | --- |
|  | (2.13) |

Where indices “***Y***” and “***Z***” in *f****Y***() and *f****Z***() implies the layers “***y***” and “***z***”, respective. Note, *f*() and *f*() are scalar activation function as usual, like sigmoid function, for instance:

Note, *f*() and *f*() are *m*x*s* matrix and *p*x*q* matrix, respectively. Derivatives of activation matrices *f*() and *f*() are determined by taking derivative of activation function on transposition of and , which are 4-dimension tensors called *Jacobian tensors* *f*’(), *f*’().

|  |  |
| --- | --- |
|  | (2.14) |

|  |  |
| --- | --- |
|  | (2.15) |

Each element () of Jacobian tensors *f*’() (*f*’()) is a *s*x*m* (*q*x*p*) *gradient matrix* ∇*f*(*x*’*ij*) (∇*f*(*y*’*ij*)).

Where,

Therefore, each gradient matrix or has only one element which can be non-zero whereas other remaining elements are zero. Note, *f*’() and *f*’() are first-order derivatives of scalar activation function as usual, like derivative of sigmoid function, for instance:

Because determining 4-dimension Jacobian tensors Jacobian tensors *f*’(), *f*’() is hazard, vectorization technique is applied into transforms *f*’() and *f*’() into vectors. Given *n*x*r* matrix ***X***, *vectorization operator* on ***X*** denoted vec(***X***) is the concatenation columns of ***X*** into single *nr*x1 column vector, for instance:

Note,

Of course, it is easy to transform vector vec(***X***) backward original matrix ***X***.

Therefore, Jacobian tensors *f*’() and *f*’() is vectorized into Jacobian matrices *f*’(vec()) and *f*’(vec()) whose elements are the same to elements in *f*’() and *f*’(). Although *f*’() and *f*’() are 4-dimesion tensors whereas *f*’(vec()) and *f*’(vec()) are *ms* x *ms* matrix and and *pq* x *pq* matrix because dimensions of , *f*(), , and *f*() are *m*x*s*, *m*x*s*, *p*x*q*, and *p*x*q*, it is possible to identify *f*’() with *f*’(vec()) and identify *f*’() with *f*’(vec()) because *vectorization technique* conserves all features of *f*’() and *f*’().

|  |  |
| --- | --- |
|  | (2.16) |

*Feedforward network* (*FNN*) with matrix layers represented by a series of layers ***X***0, ***X***1, ***X***2,…, ***X****K* is called *K*-layer FNN which is represented as follows:

By extending notation of ANN with matrix operators, how to learn ANN by *stochastic gradient descent* (SGD) algorithm associated *backpropagation algorithm* is not so difficult to be extended for estimating weight matrices *U*, *V* and bias matrices Θ, Φ.

Within backpropagation algorithm, *U*, *V*, and Φ are estimated before *P*, *Q*, and Θ because matrices ***X***, ***Y***, and ***Z*** are input layer, hidden layer, and output layer, respectively without loss of generality. In general case, likelihood of ANN denoted *l*(***Z*** | *P*, *Q*, Θ, *U*, *V*, Φ) is defined so as to estimate *P*, *Q*, Θ, *U*, *V*, and Φ by maximizing *l*(***Z*** | *P*, *Q*, Θ, *U*, *V*, Φ) with SGD when *l*(***Z*** | *P*, *Q*, Θ, *U*, *V*, Φ) is function of *P*, *Q*, Θ, *U*, *V*, and Φ given data ***Z***. Let the dot “*.*” denote the set of parameters {*P*, *Q*, Θ, *U*, *V*, and Φ} and so, let *l*(***Z*** | *P*, *Q*, Θ, *U*, *V*, and Φ) = *l*(***Z*** | *.*) = *l*(***Z***)

Which implies within context of SGD:

Note, ∇*Pl*(***Z***), ∇*Ql*(***Z***), ∇Φ*l*(***Z***), ∇*Ul*(***Z***), ∇*Vl*(***Z***), and ∇Φ*l*(***Z***) are gradients (first-order derivatives) of likelihood *l*(***Z***) with regard to *P*, *Q*, Θ, *U*, *V*, and Φ, respectively whereas *γ* (0 < *γ* ≤ 1) is learning rate and the superscript “*T*” denotes transposition operator of matrix and vector. Therefore, the main problem which is solved is how to calculate these gradients.

As usual, *likelihood* *l*(***Z***) is defined as the negative of squared *Frobenius norm* given the real output ***Z***’.

|  |  |
| --- | --- |
|  | (2.17) |

Where,

Because high dimensions of gradients related to tensors, vectorization technique is applied into calculating these gradients. Let ∇***Z****l*(***Z***) be the derivative of likelihood with regard to ***Z***:

By looking appendix within backpropagation algorithm, gradients ∇*Vl*(***Z***), ∇*Vl*(***Z***) and ∇Φ*l*(***Z***) are determined as follows:

Where *r*(*U*) is the number of rows of parametric matrix *U* and *Ir*(*U*) is the square identity matrix whose dimension is *r*(*U*) x *r*(*U*). Similarly, *c*(*V*) is the number of columns of parametric matrix *V* and *Ic*(*V*) is the square identity matrix whose dimension is *c*(*V*) x *c*(*V*).

Let ∇***Y****l*(***Z***) be the derivative of likelihood with regard to ***X***, which is determined by looking up appendix as follows:

Consequently, gradients ∇*UL*(*P* | ***Z***) and ∇Θ*L*(*P* | ***Z***) are determined as follows:

Where *r*(*P*) is the number of rows of parametric matrix *P* and *Ir*(*P*) is the square identity matrix whose dimension is *r*(*P*) x *r*(*P*). Similarly, *c*(*Q*) is the number of columns of parametric matrix *Q* and *Ic*(*Q*) is the square identity matrix whose dimension is *c*(*Q*) x *c*(*Q*).

As a summary, we have:

|  |  |
| --- | --- |
|  | (2.18) |

Note that *Ir*(*U*), *Ic*(*V*), *Ir*(*P*), and *Ic*(*Q*) are identity matrices whose dimensions are determined in their contexts.

Moreover, the binary operator denotes *Kronecker product*. Given two matrices *A* and *B* whose dimensions are *m*x*n* and *p*x*q*.

Their Kronecker product is (Wikipedia 2025):

Dimension of matrix *AB* is *mp* x *nq*.

Please pay attention that ∇*l*(vec(.)) are column vectors here for easy computation although they are row vectors according to numerator layout convention because they are gradients of scalar-by-matrix function *l*(vec(.).

As a result, parameters *P*, *Q*, Θ, *U*, *V*, and Φ are estimated by association of SGD, backpropagation algorithm, and vectorization technique:

Of course, it is easy to transform vector parameter vectors vec(*P*), vec(*Q*), vec(Φ), vec(*U*), vec(*V*), and vec(Θ) backward original matrices *P*, *Q*, Θ, *U*, *V*, and Φ, respectively.

In the general case of feedforward network (FNN) with matrix layers represented by a series of layers ***X***0, ***X***1, ***X***2,…, ***X****K* is called *K*-layer FNN as follows:

|  |  |
| --- | --- |
|  | (2.19) |

It is easy to extended estimation equations of weight parameters *Uk*, *Vk* and bias parameter Θ*k* given likelihood *l* = *l*(***X****K*). For *k* = *K* at the last layer (output layer) given real output ***X***’*K* from environment, we have:

|  |  |
| --- | --- |
|  | (2.20) |

Such that:

|  |  |
| --- | --- |
|  | (2.21) |

For all remaining *k* (s) such that 1 ≤ *k* ≤ *K*–1 at all hidden layers, we have:

|  |  |
| --- | --- |
|  | (2.22) |

Such that:

|  |  |
| --- | --- |
|  | (2.23) |

Where *γ* (0 < *γ* ≤ 1) is learning rate. Please pay attention that functions *r*(*A*) and *c*(*A*) returns the number of rows and the number of columns for matrix *A*. The subscripts ***X****k* in the derivatives implies the layer ***X****k* of function *f*(.). If *l*(.) is likelihood function, the gradient is called *reward* at layer ***X****k* and its negative is called *error* at layer ***X****k*. In general, learning parameters of artificial neural network (ANN) in the most general case of matrix data is summarized succinctly as follows:

If *Uk* is ignored, it is identity matrix . If *Vk* is ignored, it is identity matrix .

Vectorization technique causes a boom of computational tasks related to taking derivatives. Fortunately, because the activation function *f*(.) is defined particularly on every matrix element so that is diagonal matrix whose all elements outside diagonal are zeros, the multiplication of the derivative *f*’(vec(.)) within vectorization context is wise-multiplication for element-by-element multiplication. For instance:

Where the notation denotes wise-multiplication, for example:

Please pay attention that is not the derivative of the activation function *f*(.) with respective to , indeed, which is the matrix of taking derivatives of all elements of , which means that:

It is interesting that vectorization technique is not essential for learning parameters due to support of wise-multiplication, which is easy to implement by programming techniques.

The feedforward network (FNN) with matrix layers is the most general case but it is not usual case. As aforementioned, in the usual case of feedforward network (FNN) with vector layers represented by a series of layers ***x***0, ***x***1, ***x***2,…, ***x****K* is called *K*-layer FNN as follows:

As usual, how to train FNN is association of stochastic descent gradient (SGD) and backpropagation algorithm, which in turn focuses on general maximizing likelihood function which is equivalent to minimizing error function, which in turn focuses on how to determine gradients of likelihood function (error function) such that weight *Wk* and bias Θ*k* at layer *k* for all *k* from *K* back to 1 are estimated iteratively as follows:

Where and are gradients of likelihood function with respect to *Wk* and Θ*k*, respectively. It is interesting that is calculated based on and the output ***x****k*–1 of previous layer *k*–1.

Therefore, the gradient which is called error at layer *k* is most important to train FNN. It is estimated in two cases in which the first case is that is the error at the last layer (output layer) *K* whereas is the error at the layer *k* from *K*–1 back to 1.

In general, the association of SGD and backpropagation for training FNN in vector context is called vector technique:

The error or reward above is column vector for easy explanation but it is row vector according to numerator layout convention as follows:

In practical and traditional technique as aforementioned, SGD is partitioned into every neuron *xki* of layer ***x****k*, called scalar technique which is the same to the vector technique above. As a convention, the *i*th neuron ***x****k*[*i*] at the current layer *k* has input vector ***x****k*–1 at previous layer *k*–1, intermediate input scalar , and output ***x****k*[*i*] whose parameters are weight vector ***w****k*[*i*] and bias scalar Θ*k*[*i*]. Therefore, layer *k* is represented by the triple of ***w****k*[*i*], Θ*k*[*i*], and ***x****k*[*i*], such that:

Note, ***w****k*[*i*] which is the column vector includes all weights from the neurons at previous layer *k*–1 to the current layer ***x****k*[*i*].Please pay attention, let which is the column vector includes all weights from the current neuron ***x****k*[*i*] to the neurons at next layer *k*+1. It is easy to recognize that***w****k*[*i*] and are the *i*th row and the *i*th column of weight matrix *Wk*+1. Scalar technique which is the association of SGD and backpropagation for training FNN in scalar context is specified as follows:

Where error vector is column vector includes all errors of neurons *j* at the next layer *k*+1. Note, the index [*h*] indicates the *h*th neuron at the previous layer *k*–1.

Anyhow, the key point of any training method is to specified the last bias (last error) :

Or

Which depends on particular application. As usual, the last bias is equivalent to differential *dl*(***x***) of optimization function *l*(***x***) with respect to ***x*** where *l*(***x***) is function of the last layer ***x*** = ***x****K* (neuron at the last layer), according to numerator layout convention:

|  |  |
| --- | --- |
|  | (2.24) |

Note, the derivative is optional, which can be omitted if activation function at the last layer is replaced by the *optimization function* *l*(***x***), but is often derivative of activation function because the optimization function *l*(***x***) is often separated from activation function *f*(***x***). In general, the last bias is product of activation derivative and *optimization derivative* *l*’(***x****K*). If optimization function replaces activation function, then the *last bias* is optimization derivative. Note, now is row vector according to numerator layout convention. If *l*(***x***) is likelihood function, the optimization problem is *maximization problem*. If *l*(***x***) is error function, the optimization is *minimization problem*. As usual, *l*(***x***) is defined as the negative of squared Frobenius norm of two matrices (vectors), which derives the result above. Let *b*(***x***) be the *core* *last bias*, which is often row vector that is the gradient of maximization function or the negative of the gradient of minimization function because stochastic gradient descent (SGD) algorithm pushes the concerned maximizer (minimizer) moved along in the same direction (opposite direction) to gradient of the maximization function (minimization function) multiplied with learning rate. In general, the core last bias is the optimization derivative or optimization gradient.

|  |  |
| --- | --- |
|  | (2.25) |

Such that the entire last bias is:

Where *ε*(***x***) is *error function* which is the negative of likelihood function in definition. Most of training artificial neural network (ANN) focuses on how to calculate the core bias *b*(***x***), which relates to how to define the likelihood function or the error function. Note, error function is also called *loss function*. In general, likelihood function and error function (loss function) are mutually opposite but both of them are optimization function which is optimized by SGD so as to estimate the last parameters *WK* and Θ*K* which in turn are the base of estimating other parameters at previous layer in the backward direction from *K* back to 1, which is the true meaning of association of stochastic gradient descent (SGD) and backward algorithm. Anyhow, calculating the core last bias *b*(***x***) which is derivative of likelihood function / error function is the core of training artificial neural network (ANN). As usual, *b*(***x***) is row vector according to formal convention.

Note, because *b*(***x***) is the core last bias at the last layer (output layer) ***x****K*, it is possible to concern that:

Let be the bias at the *k*th layer where 1 ≤ *k* ≤ *K*–1, which is propagated backward from the core last bias with note that these biases are column vectors for easily understandable explanation:

|  |  |
| --- | --- |
|  | (2.26) |

In the most general case of matrix neural network, the core last bias at the *k*th layer is extended as follows:

Now we research how to train artificial neural network (ANN) for *classification task* where minimization function (error function) is defined as *entropy loss* with *soft-max function*. Given vector sample ***x*** = ***x****K* = (*x*1, *x*2,…, *xn*)*T* representing the last layer *K* (output layer *K*) is translated into probabilistic vector because each output neuron *xi* belonging to output layer ***x*** corresponds to the probability , for instance, is the probability of the class *i* represented by value *xi* in classification task. As a convention, *xi* is called *class value* and is called *class probability*. Therefore, *ANN classifier* is the feedforward ANN whose output layer ***x*** = ***x****K*is translated into the probabilistic vector of *n* classes.

As usual, soft-max function is applied into translating class value *xi* to class probability .

Therefore, we obtain the following convention:

Such that *xi* are also random variables and are their probabilities representing distribution *P*(*x*) whereas ***x*** = (*x*1, *x*2,…, *xn*)*T* is *sample*. Soft-max function and class probability will be mentioned again later. ANN classifier is based on definition of *cross-entropy loss function* as minimization function and so, it is necessary to skim over concept of entropy and cross-entropy. Given probability representing any class probability measures the certainty of a random variable *x* representing any class value variable, which causes an opposite concept that is called *Shannon information* of *x* or *information* of *x*, in short, denoted *I*(*x*). Therefore, *I*(*x*) measures the uncertainty of random variable, in which, the larger the information *I*(*x*) is, the more uncertain the random variable *x* is, the more information the random variable *x* reveals.

Which implies:

Given output vector ***x*** = ***x****K* = (*x*1, *x*2,…, *xn*)*T* whose respective probabilities represent distribution of *x* so that *entropy* of random variable *x* is defined as expectation of the information *I*(*x*) given the distribution .

The entropy *H*(*x*) measures the dynamics of random variable *x* with respective to the probability of such random variable *x*, which is average value or representing value of the information *I*(*x*) in general viewpoint. In the opposite, the quality of classification task aims to minimize the degree of such dynamics and so, entropy *H*(*x*) is the base of cross-entropy loss function. Given random variables *x* and *y*, *cross-entropy* of *x* given *y* is expectation of the information of *x* over distribution of *y*, which is specified as follows:

Where *x* and *y* are predicted random variable and real random variable, respectively. The meaning of cross-entropy is the same to normal entropy, which implies uncertainty, but cross-entropy establishes the relationship between two random variables in which the dynamics of random variable is aligned. When sample ***x*** represents the output layer *K* and sample ***y*** = ***x***’ = ***x****K*’ = (*x*1’, *x*2’,…, *x n*’)*T* represents real values of the output layer *K* from environment, the cross-entropy of computed output values and real output values is re-written as follows:

Note, *P*(*xi*’) is probability of real random variable *xi*’:

Such that:

As a convention, *xi* is called *predicted* (*class*) *value*, *xi*’ is called *real* (*class*) *value*, is called *predicted* (*class*) *probability*, and *pi* is called *real* (*class*) *probability* whereas ***x*** and ***x***’ are called *predicted* (*class*) *value sample* and *real* (*class*) *value sample*, respectively while and ***p*** are called *predicted* (*class*) *distribution* and *real* (*class*) *distribution*, respectively. Besides, *x* and *x*’ are called *predicted* (*class*) *random variable* and *real* (*class*) *random variable*, respectively, representing any predicted class value *xi* and any real class value *xi*’, respectively. For *multi-classification* by ANN classifier, given an input vector, predicted value sample ***x*** and predicted distribution are determined, which predicted probabilities (s) are larger than 0.5 then the target object associated with the input vector belongs to such class *i* (s). Of course, an object can belong to more than one class. Now the most important problem is to train ANN classifier where the quality of classification task aims to minimize the degree of dynamics of real value sample ***x***’ and so, the entropy *H*(*x*’, *x*) is considered as the cross-entropy loss function loss(***x***) which will be minimized.

|  |  |
| --- | --- |
|  | (2.27) |

Where,

|  |  |
| --- | --- |
|  | (2.28) |

Such that the core last bias *b*(***x***) is the negative of gradient of loss(***x***) with respect to ***x*** with suppose that loss(***x***) is convex function.

Please pay attention that the real distribution ***p*** = (*p*1, *p*2,…, *pn*)*T* is taken from environment with note that common classification applications let *pi*=1 if an target object belongs to class *i* and otherwise, the applications let *pi*=0 if the target object does not belong to class *i*. In the case that, there is no real distribution ***p*** but there exists real sample ***x***’ = ***x***’*K* = (*x*’1, *x*’2,…, *x*’*n*)*T*, then each real probability *pi* can be calculated by soft-max function given real value *xi*’ as follows:

Recall that the core last bias *b*(***x***) is the negative of gradient of the cross-entropy loss function loss(***x***) with respect to ***x***.

Such that the lass bias for training ANN classifier is determined as follows:

Therefore, how to calculate the cross-entropy gradient ∇loss(***x***) becomes the main problem of training ANN classifier. Indeed, ∇loss(***x***) is the following row vector:

Which means that ∇loss(***x***) is the row vector of *n* partial derivatives where is the partial derivative of loss(***x***) with respect to *xj* as follows:

Where is derivative of real probability with respect to *xj* as follows:

If *i*=*j*, we have:

If *i*≠*j*, we have:

This implies the derivative of real probability with respect to *xj­* is specified as follows:

Where,

Therefore, the partial derivative of loss(***x***) with respect to *xj* is determined as follows:

As a result, cross-entropy gradient ∇loss(***x***) is totally determined as follows:

|  |  |
| --- | --- |
|  | (2.29) |

Where,

Therefore, the lass bias for training ANN classifier is totally determined:

Such that all parameters of ANN classifier are totally estimated by association of stochastic gradient descent (SGD) algorithm and backpropagation algorithm.

Such that:

Please pay attention that ∇loss(***x***), , and now are row vectors according to numerator layout convention. In the most general case of matrix neural network where layers are matrices including the cross-entropy gradient ∇loss(***x***) as matrix, then:

If ∇loss(***x***) is matrix, its rows or columns will be determined by the same way of cross-entropy minimization method aforementioned.

# 3. Convergence of learning algorithm

Recall that there are two rules for learning neural network (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 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, 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 study 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 the 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 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. Conversely, SGD is developed from delta rule in practice. 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 which is stochastic approximation of gradient descent (GD) method. Given target function *f*(***w***), GD is an iterative algorithm that moves the parameter ***w*** along descending direction which is the opposite of gradient of *f*(***w***) at every time point (or iteration) *t* until reaching the optimizer ***w***\*.

Note, *γt* is 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:

|  |  |
| --- | --- |
|  | (3.1) |

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

|  |  |
| --- | --- |
|  | (3.2) |

If *n* is too large for a very complicated gradient ∇*f*(***w***) to be calculated at one time then, SGD is a variant of GD 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***.

|  |  |
| --- | --- |
|  | (3.3) |

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. Because is selected among *n* member functions *fi*(***w***), probability distribution of is even as follows (De Sa, 2021, p. 2):

This probability distribution 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 any two vector ***w*1** and ***w***2 we have (Wikipedia, Lipschitz continuity, 2001):

Where *Li* 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 ***w*** is zero vector, we have:

It is possible to understand commonly thatLipschitz continuous function does not increase/decrease unexpectedly as well as does not have singularity point, for instance, Lipschitz continuous function does not have saddle point. Therefore, convex/concave function is Lipschitz continuous, obviously when convexity and concavity are stronger than Lipschitz continuity if function is continuous.

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, which is stated equivalently that every member function *fi*(***w***) and its gradient are Lipschitz continuous, because 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). That gradient of every member function *fi*(***w***) must be bounded means every member function *fi*(***w***) must be Lipschitz continuous too:

|  |  |
| --- | --- |
|  | (3.4) |

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:

Recall that 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 within bounded Lipschitz continuous gradient condition, its derivative ∇2*fi*(***w***) which is Hessian matrix (second-order derivative) of *fi*(***w***) is bounded according to the important property, as follows:

|  |  |
| --- | --- |
|  | (3.5) |

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. Besides, according to such important property, the bounded Lipschitz continuous gradient condition is equal to the condition that *all* *fi*(***w***) *and their gradients* ∇*fi*(***w***) *are Lipschitz continuous*. The bounding of ∇2*fi*(***w***) as ||∇2*fi*(***w***)|| ≤ *Hi* 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 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. In other words, here ∇2*fi*(***w***) shrinks vector space. Obviously, we also have:

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

Where let,

Recall that SGD is an 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 norm 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 associated with the stochastic distribution *g*(***w****t*) and selective distribution *P*(). In general, we will prove the equation as follows:

|  |  |
| --- | --- |
|  | (3.6) |

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 can 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):

|  |  |
| --- | --- |
|  | (3.7) |

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):

|  |  |
| --- | --- |
|  | (3.8) |

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

We have:

|  |  |
| --- | --- |
|  | (3.9) |

Due to:

We obtain:

As a result, we assert that SGD will converge if all member functions *fi*(***w***) and their gradients ∇2*fi*(***w***) are 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 because squared error function and its gradient are Lipschitz continuous. The condition of decreased learning rate is not hazard by setting it to be inversely proportional to time point. In other words, the convergence of delta rule is asserted with Lipschitz continuity.

# 4. Recurrent network

Default NN is feedforward NN in which there is no circle in the network, which means that there is no feedback connection from next layers back to previous layers. Conversely, *recurrent neural network* (*RNN*) (Kröse & Smagt, 1996, p. 47) allows such feedback connection, which means that an output unit or hidden unit can connect to a previous hidden unit directly or indirectly. Because input layer is fixed or not counted in the network, feedback connections exist among only hidden units and output units. In general, there are two types of feedback 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 feedback connection. Moreover, as usual immediate units connect to 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 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 backwards 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 feedback connection and the extra input units are called state units, as follows (Kröse & Smagt, 1996, p. 48):

A picture containing sketch, diagram, drawing, line

Description automatically generated

**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 backwards 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 feedback connection and the extra input units are called context units, as follows (Kröse & Smagt, 1996, p. 49):

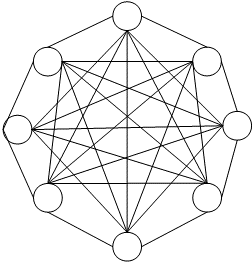
A picture containing text, line, diagram, font

Description automatically generated

**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 feedback connections between hidden units and extra input units whereas Jordan network makes feedback 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 feedback connections where each feedback 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):

|  |  |
| --- | --- |
|  | (4.1) |

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:

|  |  |
| --- | --- |
|  | (4.2) |

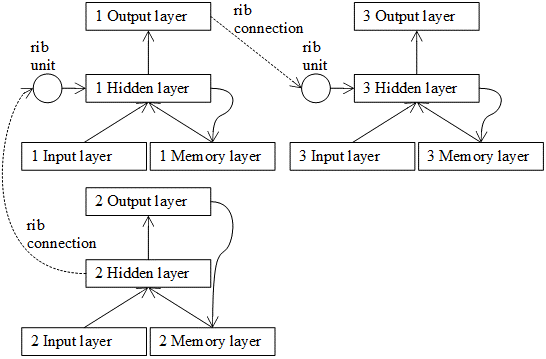
If time point is not concerned, a unit *k* is stable if its *yk* is not changed from the previous value.

At the time Hopfield network was invented, it was used to model associative memory, which means that after its weights are trained from sample, units can become stable as persistent memory. Therefore, given a input vector ***x*** = (*x*1, *x*2,…, *xn*)*T*, after applying Hebbian rule many times, the associative memory can be reached at which all units are stable, which can be considered as training process of Hopfield network.

|  |
| --- |
| *Input*: input vector ***x*** = (*x*1, *x*2,…, *xn*)*T* of *n* units, weight matrix *W* is initialized arbitrarily with suppose *W* is symmetric, and bias vector Θ is initialized as zero vector Θ = **0***T*.  *Output*: weight matrix *W* and biases vector Θ are trained at which all units are stable.  All outputs are initialized by inputs such that *yk* = *xk* for all *k*.  Repeat  Calculate biases *θk* and outputs *yk* of all units according to Bruce algorithm (Kröse & Smagt, 1996, p. 52) and propagation rule as follows:  For every pair of two units *j* and *k* where *j* ≠ *k*, their weight *wjk* are updated according to Hebbian rule as follows:  Until all units are stable |

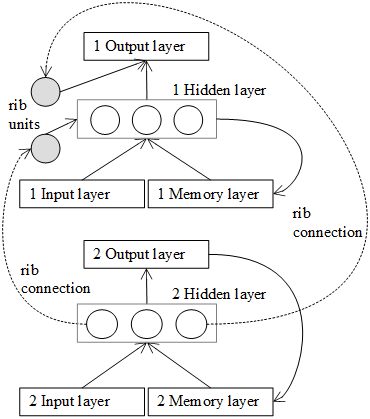
**Table 3.1.** Learning Hopfield network

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 feedback 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.

# 5. Self-organizing network

Standard feedforward neural network (feedforward NN) as well as recurrent neural network (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 an 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. A simple competitive SON is a feedforward NN having two layers in which all input units *i* connect to all output unit *o* where given input vector ***x*** = (*xi*) there is only one output unit *o* is valid, which is called activated output unit or *winner* (Kröse & Smagt, 1996, pp. 57-58).

A picture containing line, diagram, circle, origami

Description automatically generated

**Figure 4.1.** Simple network of competitive learning

The winner can be considered as cluster if competitive SON aims to clustering data. There are two methods for winner selection such as dot product method and Euclidean distance method. According to dot product method, because the bias is assumed to be 0, propagation rule becomes dot product as follows (Kröse & Smagt, 1996, p. 58):

|  |  |
| --- | --- |
|  | (5.1) |

Where ***x*** = (*xi*) = (*x*1, *x*2,…, *xn*,…)*T* is input vector and ***w****o* = (*w*1*o*, *w*2*o*,…, *wno*,…)*T* whereas *yo* is output of output unit *o*. Note, activation function *f*(.) is not applied to this competitive learning. The winner *o* is the output unit *o* whose output is maximum (Kröse & Smagt, 1996, p. 58).

|  |  |
| --- | --- |
|  | (5.2) |

After the winner was selected, its output is activated to be zero as *yo* = 1 and other outputs of output units are deactivated to be zero as = 0 (Kröse & Smagt, 1996, p. 58).

|  |  |
| --- | --- |
|  | (5.3) |

Within dot product method, only weight vector ***w****o* = (*w*1*o*, *w*2*o*,…, *wno*,…)*T* of the winner *o* is updated to be moved forward the input vector ***x*** and then normalized, as follows (Kröse & Smagt, 1996, p. 58):

|  |  |
| --- | --- |
|  | (5.4) |

The denominator of equation above is used to normalize the winner weight vector ***w****o* where notation ||.|| denotes Euclidean norm. Note, *γ* (0 < *γ* ≤ 1) is learning rate as usual.

Similarly, Euclidean distance method selects the winner based on Euclidean distance between output weight vector and input vector. Therefore, the winner *o* is the output unit *o* that Euclidean distance between the output weight vector ***w****o* and the input vector ***x*** is minimum, which means that the winner *o* is the nearest to the input vector ***x***.

|  |  |
| --- | --- |
|  | (5.5) |

After the winner was selected, its output is activated to be zero as *yo* = 1 and other outputs of output units are deactivated to be zero as = 0.

Like dot product method, only weight vector ***w****o* = (*w*1*o*, *w*2*o*,…, *wno*,…)*T* of the winner *o* is updated to be moved forward the input vector ***x*** but such winner weight vector is often not normalized.

|  |  |
| --- | --- |
|  | (5.6) |

Note, *γ* (0 < *γ* < 1) is learning rate as usual. Indeed, the winner weight vector updating conforms to delta rule. Indeed, the squared error of output unit *o* is:

|  |  |
| --- | --- |
|  | (5.7) |

Gradient of the squared error of output unit *o* with regard to *wio*, known as tangent vector of ***ε***(***w****o*), is:

|  |  |
| --- | --- |
|  | (5.8) |

Note, is Jacobian matrix but the equation above expresses tangent vector for easily understandable explanation.

Obviously, the rule of updating winner weight vector ***w****o* = ***w****o* + *γ*(***x*** – ***w****o*) is result of stochastic gradient descent (SGD) method and so, its convergence is asserted as same as the theorem is stated in (Kröse & Smagt, 1996, p. 60). However, there is a question that how the error between output unit *o* and input unit *i* is defined as ½(*wio* - *xi*)2 rather than ½(*wioxo* – *xi*)2. Exactly, the error is ½(*wioxo* – *xi*)2 but *xo* is assumed to be 1 as *xo* = *yo* = 1 because the output unit *o* is assumed to be the winner and hence, we have ½(*wioxo* – *xi*)2 = ½(*wio*\*1 – *xi*)2 = ½(*wio* - *xi*)2. Competitive SON can be extended with many layers, which is learned by backpropagation algorithm based on SGD without modification.

Kohonen network is an extension of competitive SON, in which outputs of output units are ordered. For instance if input vector ***x*** = (*x*1, *x*2,…, *xi*,…, *xm*) is a vector in real vector space ℝ*m* and output vector *y* = (*y*1, *y*2,…, *yo*,…, *yn*) is a vector in real vector space ℝ*n*, there are some orderings which are defined in ℝ*m* and ℝ*n*. Based on such orderings, the concept of neighborhood is defined. Given two output units *o* and *o*’, a so-called neighborhood function *g*(*o*, *o*’) is defined so that it should be inversely proportional to distance between *o* and *o*’. For example, *g*(*o*, *o*’) is defined based on exponential function as follows:

|  |  |
| --- | --- |
|  | (5.9) |

Note, *g*(*o*, *o*) or g(*o*’, *o*’) is always 1 regardless of how to define *g*(*o*, *o*’). Two output units *o* and *o*’ are neighbors together if their neighborhood function *g*(*o*, *o*’) is large enough (larger than a threshold) or their distance is small enough (smaller than a threshold). Winner selection methods such as dot product method and Euclidean distance method are still applied into Kohonen network but the rule of updating winner weight vector is extended to neighbors of the winner unit *o*. Concretely, for the winner *o*, we still have:

For any other output units *o*’ which are neighbors of the winner *o*, their weight vector is updated as follows:

|  |  |
| --- | --- |
|  | (5.10) |

Note, *nb*(*o*) is a set of units which are neighbors of the winner *o* where the neighborhood is determined based on neighborhood function *g*(*o*, *o*’) or Euclidean distance. Kohonen network can be extended with many layers, which is learned by backpropagation algorithm based on SGD without modification except that putting neighborhood function *g*(*o*, *o*’) into the updating rule of output units as the equation above.

# 6. Reinforcement learning

Recall that there are three main types of machine learning where machine learning is a branch of artificial intelligence (AI):

* Supervised learning matches inputs and outputs to find out rules and knowledge where the outputs direct such knowledge searching. Classification is a popular supervised learning algorithm.
* Unsupervised learning analyzes inputs so as to discover patterns under the inputs. Such patterns do not obey any output because simply there is no output in unsupervised learning. Clustering is a popular unsupervised learning algorithm.
* Reinforcement learning (RL) draws and finetunes adaptively and progressively rules and knowledges from environment. Control theory, game theory, robotics applications are typical examples of RL.

Neural network (NN) supports all three main types of machine learning where feedforward NN supports supervised learning and self-organizing network supports unsupervised learning, which is mentioned in previous sections. Fortunately, NN also supports RL where concepts and algorithms of RL are implemented in NN. Therefore, we should skim what RL is. In general, RL has two main objects such as an *agent* and an *environment*. When the environment issues a *state*, the agent will make an *action* that responds to such state and then, the environment gives feedback to the agent by a *reward* as benefit or penalty for the agent’s action (Chandrakant, 2023). The purpose of RL is to maximize the reward such that the agent’s action is most appropriate to the environment’s state; in other words, RL maximizes the benefit of action given state. The mapping between state and action is called *policy* and so, essentially, RL finds out optimal policy. This interaction of agent and environment repeats progressively until the optimal policy is reached. The following figures (Chandrakant, 2023) sketches RL.

A picture containing text, diagram, font, line

Description automatically generated

**Figure 5.1.** Overview of reinforcement learning

There are two types of RL such as model-based RL and model-free RL (Chandrakant, 2023). As the hint of these names, model-based RL (Chandrakant, 2023) uses explicitly some mathematical model to interpret and explain RL shown by the overview figure above whereas model-free RL (Chandrakant, 2023) takes advantages of experiences to simulate the interaction between agent and environment when mathematical model is unknown or not supported. We research model-based RL first and model-free RL later. Therefore, Markov decision process (MDP) is a popular mathematical model which is applied into explaining and implementing model-based RL. MDP uses some results from dynamic programming (Wikipedia, Dynamic programming, 2002) for maximizing value function which is cumulative reward in essentially besides taking advantages of Markov property that the probability of future state depends only on current state. So, the environment in MDP follows Markov property. The following figures sketches RL and MDP.

A picture containing text, font, line, screenshot

Description automatically generated

**Figure 5.2.** Roadmap of RL methodologies

From the figure above, this section mentions MDP because MDP is the most popular mathematical model for RL. An MDP (Wikipedia, Markov decision process, 2004) consists of 4 main components as follows (Wikipedia, Reinforcement learning, 2002):

* Let *S* be a set of states of environment and let *s* be any state belonging to *S*. Let *st* be the state at time point *t*.
* Let *A* be a set of actions of agent and let *a* be any action belonging to *A*. Let *at* be the action at time point *t*.
* Let *Pa*(*s*, *s*’) = *P*(*st*+1 = *s*’ | *st* = *s*, *at* = *a*) be the transition probability at time point *t* from the current state *st* = *s* to the next state *st*+1 = *s*’ given action *at* = *a*. This transition probability is conditional probability. A set of all transition probabilities for all states given an action compose a transition probability matrix *Pa*. The transition probability implies that Markov property where the probability of next state *s*’ depends only on current state *s*. The sum of all transition probabilities given a fixed state is 1 as .
* Let *Ra*(*s*, *s*’) be the immediate reward that the environment issues immediately when the agent does the current action *at* = *a* such that the current state *st* = *s* is changed immediately to the next state *st*+1 = *s*’. Reward function is the heart of model-based RL.

From the MDP model, the mapping from state to action is called policy which is modeled by a so-called policy function *a* = *π*(*s*). The essence of MDP is to train policy function *a* = *π*(*s*) to be optimal, which in turn maximizes a so-called *value function* based on the immediate reward function *Ra*(*s*, *s*’) which is a component of MDP. Note, maximization of value function is derived from dynamic programming. For any state *s*, value function *V*(s) is expectation of reward function *Ra*(*s*, *s*’) multiplied with discount factor *αt* under the transition distribution *Pa*(*s*, *s*’). Therefore, *V*(s) is also called *discounted reward expectation*, *expected cumulative reward,* or *expected reward*, which is determined from *s* = at some *tk*th time point to infinity.

|  |  |
| --- | --- |
|  | (6.1) |

Where,

Proof,

(Due to Markov property)

The discount factor *αt* (0 < *αt* ≤ 1) indicates that a reward can be delayed at time point *t*, which implies that discount factor indicates the importance of future reward to immediate reward. The longer the delay is, the smaller discount factor is and so, only the first reward gains highest discount factor . If = 1 then, the first reward is immediate reward such that which is reserved. Discount factor should be inversely proportional to time point, for example *αt* = 1 / (t+1). The equation above is the general case of value function with infinite expectation. *Dynamic programming* solves problem of MDP for finding optimal policy by firstly, redefining value function *V*(*s*) recursively as follows (Wikipedia, Markov decision process, 2004):

|  |  |
| --- | --- |
|  | (6.2) |

Now value function is determined by a finite sum and so, it is also called *discounted reward sum* or *practical expected reward* in which *s* *S*, *a* *A*, and both *S* and *A* are finite sets. Moreover, *s*’ belongs to the set *A*(*s*) which contains all next states of state *s*. The equation above is also called Bellman expectation equation. Indeed, value function here is an estimate of discount reward expectation. In first view, discount factor *α* (0 < *α* ≤ 1) is fixed but, actually, it is decreased in time because of the recursion inside the formulation of finite *V*(*s*) and hence, only the immediate rewards *Rπ*(*s*)(*s*, *s*’) are reserved. Following is the proof of *discounted reward sum* based on discount reward expectation.

Indeed, discount reward expectation called theoretical value function is the expectation based on policy *a* = *π*(*s*) via transition probability.

For simplicity, only one discount factor *α* which is made power is used and policy *a* = *π*(*s*) is applied so that theoretical value function is simpler.

Theoretical value function is limited by accounting only immediate rewards from time point *t* as follows (Gemini 2025):

Let *Gt* be the total return that is sum of all future rewards from time point *t*.

As a convention, *Gt* is called *expected return* or *target value*. The limited value function is rewritten:

It is necessary to determine the conditional total return . We have:

Because *st* is fixed as *st* = *s* then *st*+1 is fixed as *st*+1 = *s*’, the conditional total return is estimated as follows:

Expectation of such conditional total return is:

Therefore, the limited value function is:

This limited value function is discounted reward sum in dynamic programming, which is estimate of discounted reward expectation in practice:

Consequently, policy function *π*(*s*) is updated as maximizer regarding value function as follows (Wikipedia, Markov decision process, 2004):

|  |  |
| --- | --- |
|  | (6.3) |

An implementation of MDP learning is an iterative algorithm so that whenever the environment feeds back a next state *st*+1 and gives back a reward for the agent’s action *at* at the current state *st* (time point *t*), the iterative algorithm will update value and policy as follows:

|  |
| --- |
| Value update rule:  Policy update rule: |

**Table 5.1.** Markov decision process learning for model-based reinforcement learning

A possible terminated condition for the iterative algorithm is that all states are stable, which means that there is no change in policy function *π*(*s*). However, RL does not require mandatorily terminated conditions because it aims to adapt to the environment. Note that all values *V*(*s*) and *Ra*(*s*, *s*’) for all *s*, *s*’, and *a* are initialized by 0. Of course, the agent’s action *at* at the current state *st* is based on the policy function *at* = *π*(*st*) where *st* is raised by the environment.

There is no problem for model-based RL with MDP but it is hazard for model-free RL where none of transition distribution and reward function is specified explicitly. Fortunately, Q-learning (Wikipedia, Q-learning, 2004) is applied into solving the lack of mathematical model in model-free RL in which there is no transition probability *Pa*(*s*, *s*’) and reward function *Ra*(*s*, *s*’). With Q-learning, model-free RL broadens its applications, especially neural network learning. At time point *t*, the environment still gives back a reward *Rt* in model-free RL but such *Rt* is only a value which is not the function *Ra*(*s*, *s*’) in model-based RL. Given time point *t*, value function *V*(*s*) in model-based RL is replaced by Q-value *Q*(*st*, *at*) for model-free RL and such Q-value is learned as follows (Wikipedia, Q-learning, 2004):

|  |  |
| --- | --- |
|  | (6.4) |

Where *γ* (0 < *γ* ≤ 1) is learning rate. The equation above is called Bellman equation. Therefore, whenever the environment feeds back a next state *st*+1 and gives back a reward *Rt* for the agent’s action *at* at the current state *st* (time point *t*). the iterative algorithm of Q-learning for model-free RL is described as follows:

|  |
| --- |
| Q-value update rule:  Policy update rule: |

**Table 5.1.** Q-learning for model-free reinforcement learning

Note that all Q-values *Q*(*s*, *a*) for all *s* and *a* are initialized by 0. A possible terminated condition for the iterative algorithm is that all states are stable, which means that there is no change in policy function *π*(*s*). Of course, the agent’s action *at* at the current state *st* is selected based on the policy function *at* = *π*(*st*) where *st* is raised by the environment.

According to (Chandrakant, 2023), when neural network (NN) is used to implement MDP, it is a feedforward NN whose input units represent environment’s states and whose output units represent agent’s actions. The number of hidden layers indicates complexity of RL with note that deep learning, which is a modern machine learning, is implemented by a NN having as many as possible hidden layers. Because a NN for RL often needs more than one hidden layer for improving accuracy of learning method with high complexity, the combination of NN and RL is often called deep reinforcement learning (DRL). There is a question why the high complexity with many hidden layers will improve the learning accuracy. The reason is that the essence of any learning NN algorithm is to make an approximation of the desire function *v*(***x***) where ***x*** is inputs, and the approximation can be represented by an estimation function *u*(***x***). Essentially, the estimation function *u*(***x***) is a nonlinear regression function because propagation rule goes through layered weights with multiplications and summing. Because the number of hidden layers is proportional to the order of the regression function *u*(***x***), increasing such order is obviously to increase the accuracy of *u*(***x***) in estimation. Therefore, deep learning and deep reinforcement learning (DRL) attracts attention of many recent researches about artificial intelligence.

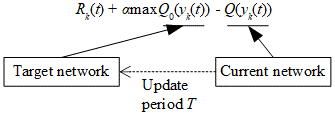
It is easier to combine NN with RL by Q-learning where inputs represent environment’s states and outputs represent agent’s actions.

Where *xk*(*t*) = *st* and *yk*(*t*) = *at* are input and output of unit *k* at time point *t*. Regarding NN, Q-value is Q-function of *xk*(*t*) and *yk*(*t*). There are two ways for coding NN for RL:

* Each input unit represents a state and each output unit represents an action. This coding is appropriate to multi-state and multi-action RL.
* Each input unit represents a possible value of state and each output unit represents a possible value of action. In this coding, inputs and outputs are binary.

Backpropagation algorithm is still valid for learning feedforward NN with Q-function. Whenever the environment feeds back a next state *st*+1 and gives back a reward *Rk*(*t*) for the agent’s action *at* = *yk*(*t*) at the current state *xk*(*t*) = *st*, the Q-function is updated as follows:

Where *α* is discount factor (0 < *α* ≤ 1) and *vk* is desired output. Note that index *k* in the maximization expression indicates browsing units in the same layer of current unit. There is a question what *Q*0(*xk*(*t*), *yk*(*t*)) is. Indeed, according to an invention of OpenAI (Choudhary, 2019), *Q*0(*xk*(*t*), *yk*(*t*)) is the function *Q*(*xk*(*t*), *yk*(*t*)) of a so-called target network which is the duplicate of current NN but parameters of target network such as weights and biases are kept intact for a period *T* of time points. After every period *T*, parameters of target networks are updated by copying from parameters of current NN. Therefore, the target network represents next states *st*+1 in Q-learning. The following figure depicts the target network for Q-learning (Choudhary, 2019).



**Figure 5.3.** Target network for Q-learning

Because *yk* is function of *xk* due to activation function *yk* = *f*(*xk*), Q-function in NN is considered as function of *yk* as follows:

The deviation of Q-function for unit *k* at time point *t* is:

If the time point *t* is implicit by default for backpropagation algorithm feeding sample time point by time point, the deviation is rewritten as follows:

|  |  |
| --- | --- |
|  | (6.5) |

Note that the expression is constant with regard to *yk*. Recall that index *k* in the maximization expression indicates browsing units in the same layer of current unit inside the target network. If there is only one unit in such layer by some specific NN coding for RL, it is possible to browse possible outputs of unit *k* inside the target network. In the equation of Δ*Q*(*yk*) above, only *Q*(*yk*) is function of *yk*. The simplest way is to set Q-function as identity function *Q*(*yk*) = *yk*. Derivative of Δ*Q*(*yk*) with regard to *xk* is:

|  |  |
| --- | --- |
|  | (6.6) |

The squared error function is square of deviation Δ*Q*(*.*). For instance, the squared error function of output unit *o* is:

|  |  |
| --- | --- |
|  | (6.7) |

The squared error function *ε*(*yh*) of hidden unit *h* is the sum of output errors *ε*(*yo*) with regard to such set of output units, as follows:

By applying stochastic gradient descend (SGD) as usual, we obtain weight update rule and bias update rule according to backpropagation algorithm, as follows:

Where,

|  |  |
| --- | --- |
|  | (6.8) |

Recall that:

Moreover, Q-functions for output units are updated by Q-learning as usual:

|  |  |
| --- | --- |
|  | (6.9) |

Indeed, Q-learning is also derived from SGD too. In NN literature, Q-function is also called the critic (Kröse & Smagt, 1996, p. 76). The sample for deep reinforcement learning with NN is {***x***(*p*), ***R***(*p*)} where input vector ***x***(*p*) is a set of states and ***R***(*p*) is a set of rewards of output units at *p* pattern. Agent’s actions are outputs *yk* from computations inside NN and next states *st*+1 are represented by the target network.

Recall that RL has two main approaches such as model-based approach and model-free approach, where Markov decision process and its dynamic programming algorithm are fundamental of model-based approach. According to Gemini 2025, model-free approach has two main methods such as value-based method focusing on value function and policy-based method focusing on policy that maps states to actions. Policy gradient (PG) method is the most popular policy-based method. The Q-learning is the most popular algorithm for value-based method in model-free approach whereas generalized advantage estimation (GAE) is a popular algorithm for policy-based method in model-free approach with note that GAE belongs to PG. GAE will be researched but it is necessary to study policy gradient method because GAE belongs to PG. Although agent and environment are not modeled with transition probability and specified immediate reward in PG, policy is specified as the probabilistic distribution *π*Θ(*a* | *s*) whose parameter is Θ called *policy parameter*, which produced the probability of taking action *a* given state *s*. Under *policy distribution* *π*Θ(*a* | *s*), a sequence of states *st*, actions *at*, and reward *Rt* is generated whenever RL model activates. Such sequence is called trajectory denoted as *τ* = (*s*0, *a*0, *r*1, *s*1, *a*1, *r2*,…, *sn*, *an*, *rn*+1, *sn*+1, …), which is considered to follows policy distribution *π*Θ(*a* | *s*) although state *s* conforms some implicit distribution like transition probability *Pa*(*s*, *s*’) aforementioned.

|  |  |
| --- | --- |
|  | (6.10) |

The purpose of policy-based method is to determine *π*Θ(*a* | *s*) by estimating Θ. We can denote to focus on policy parameter Θ because PG tries to search for optimal policy from policy distribution via maximizing objective function mentioned later.

Indeed, PG defines a so-called objective function *L*(Θ) which is expectation of expected reward over all trajectories under policy distribution, as follows:

|  |  |
| --- | --- |
|  | (6.11) |

Where *P*(*τ* | Θ) is the probability of trajectory *τ* under policy distribution *π*Θ(*a* | *s*) and *W*(*τ*) is *expected reward* which is defined based on particular methods.

As aforementioned, PG maximizes objective function *L*(Θ) to determine optimal policy represented by maximizer Θ\* of objective function *L*(Θ) and so, Θ is considered implicitly policy so that *π*Θ(*a* | *s*) is actually policy distribution, which is the reason that Θ is called parameterized policy whereas *π*Θ(*a* | *s*) is stilled called policy as usual.

By applying stochastic gradient descent (SGD) algorithm, Θ\* is estimated via some iterations by moving up according ascending gradient as follows:

Where *γ* (0 < *γ* ≤ 1) is learning rate and *L*(Θ) is gradient of objective function. Note, objective gradient ∇*L*(Θ) is the main subject of all PG methods because estimating policy parameter Θ is the ultimate purpose of policy-based method, which is not easy to be computed but fortunately, policy gradient theorem produces an excellent equation to compute such *policy gradient* though *T*(*τ*) time points of each trajectory *τ* (Gemini 2025).

|  |  |
| --- | --- |
|  | (6.12) |

Where each trajectory *τ* has *T*(*τ*) time points and *Wt* is the expected reward from time point *t*+1. As usual, *T*(*τ*) is the number of time points of trajectory *τ* = (*s*0, *a*0, *w*1, *s*1, *a*1, *w2*,…, *sT*(*τ*)–1, *aT*(*τ*)–1, *wT*(*τ*), *sT*(*τ*)). Following is the proof of policy gradient according to Gemini 2025.

Note, objective function *L*(Θ) is originally defined as the expectation over all trajectories under policy distribution:

Indeed, the probability of trajectory *τ* under policy distribution *π*Θ, which is called trajectory probability *P*(*τ* | Θ), is defined as production of transition probabilities as follows:

Where *P*(*st*+1 | *st*, *at*) is the transition probability of changing from state *st* to state *st*+1 under policy *at* and such probability is independent from policy parameter Θ. Moreover, the probability of trajectory follows Markov condition when the next state *st*+1 depends on only one current state *st*. Consequently, policy gradient is calculated as follows:

Due to derivative of logarithm function:

We obtain:

Logarithm of trajectory probability is expended as follows:

Gradient of logarithm of trajectory probability with regard to Θ is:

Therefore, gradient policy is expended as follows:

Let be the future reward of expected reward *W*(*τ*) from time point *t*0 onward, which means that is the conditional expected reward *W*(*τ* | *t*0) given time point *t*0.

Where *α* (0 < *α* ≤ 1) is discount factor. Therefore, how to determine *W*(*τ*) depends on how to determine *W*(*τ* | *t*0) which in turn depends on how to define the reward *wt*+1. Recall that *T*(*τ*) is the number of time points of trajectory *τ* = (*s*0, *a*0, *w*1, *s*1, *a*1, *w2*,…, *sT*(*τ*)–1, *aT*(*τ*)–1, *wT*(*τ*), *sT*(*τ*)) and *wt*+1 is the reward when moving from time point *t* to time point *t*+1 which defined based on particular application. As usual *wt*+1 is the immediate reward in the simplest case:

Consequently, gradient policy is re-defined by the way that we can understand that it is approximated optimally, as follows:

As a convention, *Wt* is known as expected reward, which replaces *W*(*τ*) for effective computation.

|  |  |
| --- | --- |
|  | (6.13) |

In general case, how to determine *Wt* depends on particular applications. As a result, because objective ∇*L*(Θ) is the main subject of PG, algorithms like GAE which belongs to PG aims to calculate ∇*L*(Θ) and define expected reward *Wt*. For instance, *Wt* can be state value function *Vπ*(*st*), action-state value function *Qπ*(*st*, *at*), and advantage function *Aπ*(*st*, *at*). Note, the subscript “*π*” emphasizes that such functions are under focused policy distribution *π*Θ(*a* | *s*) and thus, these function can be denoted *V*(*st*), *V*(*s*), *Q*(*st*, *at*) , *V*(*s*, *a*), *A*(*st*, *at*), and *A*(*s*, *a*) if policy distribution is default. All RL algorithms aims to maximize value function regardless of direct way or indirect way because value function implies expected reward or future reward. However, some algorithms will modify and translate value function in different forms. Particularly, PG generalize value function by expected reward. Although GAE defines expected reward *Wt* as advantage function *Aπ*(*st*, *at*), it is necessary to research state value function *Vπ*(*st*) and action-state value function *Vπ*(*st*, *at*) because *Aπ*(*st*, *at*) is based on *Vπ*(*st*) and *Vπ*(*st*, *at*). *State value function* *Vπ*(*s*) is the expected return which is summed from starting state *s*0 = *s* under policy *π*Θ.

|  |  |
| --- | --- |
|  | (6.14) |

The notation indicates that the expectation takes over all trajectories which start from state *s* (which means that the first state *s*0 at first time point is *s*) under policy *π*Θ. Recall that *T*(*τ*) is the number of time points of trajectory *τ* = (*s*0, *a*0, *r*1, *s*1, *a*1, *r2*,…, *sT*(*τ*)–1, *aT*(*τ*)–1, *rT*(*τ*), *sT*(*τ*)). Note, *rt*+1 is immediate reward from time point *t* to time point *t*+1 and *α* is discount factor (0 < *α* ≤ 1).

|  |  |
| --- | --- |
|  | (6.15) |

*Action-state value function* *Qπ*(*s*, *a*) is the expected return which is summed from starting state *s*0 = *s* with beginning action *a*0 = *a* under policy *π*Θ.

|  |  |
| --- | --- |
|  | (6.16) |

The notation indicates that the expectation takes over all trajectories which start from starting state *s* (which means that the first state *s*0 at first time point 0 is *s*) with beginning action *a* (which means that the first action *a*0 at first time point 0 is *a*) under policy *π*Θ. Action-state value function *Qπ*(*s*, *a*) is more detailed than state value function *Vπ*(*s*) because it focuses on both state *s* and action *a* while *Vπ*(*s*) is more general than *Qπ*(*s*, *a*), which means that it is possible to understand that *Vπ*(*s*) is average of *Qπ*(*s*, *a*) through many actions given only one state. Therefore, *advantage function* qualifies how much better taking the first action *a* given the first state *s* to produce a reward return is when comparing such reward return with the average return (over all actions but given the same sate *s*). Obviously, *Aπ*(*s*, *a*) is the deviation between *Qπ*(*s*, *a*) and *Vπ*(*s*).

|  |  |
| --- | --- |
|  | (6.17) |

Recall that *Vπ*(*s*) is expectation of *Qπ*(*s*, *a*) over all action:

But *Qπ*(*st*, *at*) is conditional expectation of *rt*+1 + *αVπ*(*st*+1) given *st* and *at* as follows:

|  |  |
| --- | --- |
|  | (6.18) |

Indeed, let *st* = *s*, *st*+1 = *s*’, and *a*0 = *a*, we obtain:

We have:

This implies:

Therefore,

Obviously, we obtain:

Note, both state value function *V*(*s*) and action-state value function *Q*(*s*, *a*) can be considered as expected reward as *Wt*.

Going back policy gradient which is the core of PG:

|  |  |
| --- | --- |
|  | (6.19) |

GAE defines expected reward *Wt* as advantage function:

But there may be a question that what the reason that GAE applies advantage function is. Indeed, *Wt* expresses action-state value function *Qπ*(*s*, *a*) with regard to both state and action in reasoning way but it is better if such expected return *Wt* is compared to the baseline return *b*(*st*) over all actions because the single *Qπ*(*s*, *a*) can cause high variance. In other words, subtracting *Qπ*(*s*, *a*) from *b*(*st*) can alleviate such variance, as follows:

Because state value function *Vπ*(*s*) is expectation of action-state value function *Qπ*(*s*, *a*), baseline return *b*(*st*) is estimated by *Vπ*(*s*) so that:

Obviously, *Wt* becomes advantage function so that policy gradient of GAE is:

|  |  |
| --- | --- |
|  | (6.20) |

Where,

Note, advantage function *Aπ*(*s*, *a*) is independent from policy parameter Θ, which is necessary to be estimated. Given state value function *V*(*s*), *temporal difference* (TD) error at time point *t* is defined as the deviation between expected return at time point *t* and expected return at time point *t*+1.

|  |  |
| --- | --- |
|  | (6.21) |

Where *α* is discount factor. As known that action-value function *Q*(*st*, *at*) is conditional expectation of *rt*+1 + *αV*(*st*+1) given *st* and *at* as follows:

Therefore, the sum *rt*+1 + *αV*(*st*+1) is the one-step estimate of *Q*(*st*, *at*), which leads that TD error *δt* is one-step estimate of the deviation *Q*(*st*, *at*) – *V*(*st*). In other words, TD error *δt* is one-step estimate of advantage function *A*(*st*, *at*) exact taking action at in state *st*, which implies that *δt* is the base to estimate fully *A*(*st*, *at*). The meaning of TD error is similar to the meaning of advantage function, which indicates the goodness, discrepancy, or surprise between next value in future and average value in current. Now it is necessary to estimate fully *A*(*st*, *at*). When the sum *rt*+1 + *αV*(*st*+1) is one-step return which is total discounted reward (state value function) from time point *t* considering 1 step in the future, the *k-step return* *Rt*(*k*) is defined as total discounted reward (state value function) from time point *t* considering *k* steps in the future:

|  |  |
| --- | --- |
|  | (6.22) |

Please pay attention that *V*(*st*+*k*) trajectory can be extensively sub-trajectory of *Rt*(*k*) trajectory such that starting time point of *V*(*st*+*k*) trajectory is previous to time point *t* of *Rt*(*k*). The *k-step advantage estimate* which is an approximated estimate of advantage function *A*(*st*, *at*) is the subtraction of *k*-step return *Rt*(*k*) and state value function *V*(*st*) at time point *t*.

|  |  |
| --- | --- |
|  | (6.23) |

It easy to prove that *k*-step advantage estimate is formulated by sum of TD errors as follows:

|  |  |
| --- | --- |
|  | (6.24) |

Indeed, we have:

Obviously, 1-step advantage estimate is TD error:

When *k* approaches infinity, approaches the *advantage estimate* which is real estimate of advantage function.

In practice, advantage estimate is approximated by with large enough *n* steps (*n* time points).

As a result, policy gradient of GAE is totally determined:

|  |  |
| --- | --- |
|  | (6.25) |

Where advantage function is estimated by advantage estimate .

GAE improves advantage estimate by adding a hyper-parameter *λ* (0 ≤ *λ* ≤ 1) into so as to control bias-variance trade-off of the advantage estimate:

|  |  |
| --- | --- |
|  | (6.26) |

So that:

Of course, *GAE advantage estimate* can be approximated with *n* steps (*n* time points).

|  |  |
| --- | --- |
|  | (6.27) |

Where is *n*-step GAE advantage estimate.

When GAE is implemented by feedforward network (FNN), reward function *Ra*t(*st*, *st*+1) and policy distribution *π*Θ(*at* | *st*) are coded by reward FNN *R*Ψ(*s*, *a*) and policy FNN *π*Θ(*a*, *s*), respectively. Input of reward FNN *R*Ψ(*s*, *a*) is a pair of state *s* and action *a* whereas its output is the immediate reward which is often rating that some user gives on such action and its parameter is parametric weight Ψ. Input of policy FNN *π*Θ(*a*, *s*) is a pair of state *s* and action *a* whereas its output is the probability that action *a* is taken in state *s* and its parameter is parametric weight Θ. When reward FNN *R*Ψ(*s*, *a*) is trained separately as usual, GAE is applied into training policy FNN *π*Θ(*a*, *s*). Suppose training data is split into *N* epochs and each epoch *τ* has *T* batches so that each batch *t* has a pair of states *st* and actions *at*. Policy gradient for one epoch *τ* is determined as follows:

|  |  |
| --- | --- |
|  | (6.28) |

Policy gradient for *N* epochs is determined at the same time as follows:

|  |  |
| --- | --- |
|  | (6.29) |

Where *T*(*τ*) is the number of batches belonging to epoch *τ*. Pseudo-code of GAE which trains both reward FNN *R*Ψ(*s*, *a*) and policy FNN *π*Θ(*a*, *s*) is described as follows:

Repeat whenever an epoch is fed:

Reward FNN *R*Ψ(*s*, *a*) is trained separately with the epoch.

Suppose the epoch has *T* batches so as to calculate *T* important quantities such as immediate reward function *R*Ψ(*st*), state value function *V*(*st*), *n*-step return *Rt*(*n*) and advantage function *At*.

Policy parameter Θ is estimated by policy gradient as follows:

Until some terminated conditions are met.

In general, AGE estimates reward parameter Ψ and policy parameter Θ. The terminated conditions are often the number of iterations or insignificant change in these estimates. Although action-state value function *Q*(*s*, *a*) is calculated based on outputs of reward FNN *R*Ψ(*s*, *a*), it can be modeled and trained as a separated FNN.

# 7. Conclusions

The philosophical essence of neural network (NN) is synaptic plasticity of human neuron system and the technical essence of NN is nonlinear regression mechanism by multiplicative overlap of summing weights through many layers. The perfect nonlinear regression function, which is target of NN learning, is approximated by the multiplicative overlap of applying propagation rule (being linear function if ignoring activation function) many times, which can be considered as an interpolation of the nonlinear function by many linear functions via a complex topology. The approximation will be unfeasible or ineffective unless there is support of stochastic descent gradient method. Moreover, the approximation is made smoother by activation function. This is the reason that deep learning with multiple layers will increase effectiveness and accuracy of NN because deep learning increases order of such nonlinear regression model. Moreover, the partition of NN into layers where there is an output layer implicitly reflects analytic and synthetic mechanism which is appropriate to high processing applications like image processing. The evolution of NN via Hebbian rule and delta rule learning which simulates human neuron system is appropriate to intelligent applications like control applications and game applications. In general, the ability of NN extensions is fully promising, especially NN is combined with evolutionary programming field such as genetic algorithm and social intelligence. When NN focuses on individual intelligence via human brain, there is a so-called social intelligence which is a subdomain of evolutionary programming field where social intelligence focuses on the intelligence inside a group of individuals via interactions. The combination of individual intelligence and social intelligence issues a multi-faceted overview of biological world as aforementioned in the abstract that machine learning (ML), which is a branch of artificial intelligence (AI), sets first bricks to build up an infinitely long bridge from computer to human intelligence. This great construction may be more feasible a little bit by concerning such multi-faceted biological problem when AI also computer science does not reach the limitation of approaching miracle biological phenomenon yet. Fishbone NN mentioned in this research is a theoretical trial of the combination of individual intelligence and social intelligence.

# Appendices

**A1. Derivatives of matrix function**

This appendix refers to information from Gemini (gemini.google.com). As a convention, scalar variable, vector variable, and matrix variable are often denoted as normal-lowercase letter *x*, bold-lowercase letter ***x***, and bold-uppercase ***X***, respectively whereas scalar constant and vector are denoted as uppercase letter *a* while matrix constant is often denoted as normal-uppercase *A*, by default if there is no explanation. Moreover, vector is *column vector* by default. Function *y* = *f*(*x*) whose *x* and *y* are scalar, function *y* = *f*(***x***) whose ***x*** is vector and *y* is scalar, and function *y* = *f*(***X***) whose ***X*** is matrix and *y* is scalar are called *scalar-by-scalar* function, *scalar-by-vector* function, and *scalar-by-matrix* function, respectively whose derivatives are called *derivative* *f*’(*x*), *gradient vector* ∇*f*(***x***), and *gradient matrix* ∇*f*(***X***).

Note, though:

But really:

Due to *numerator layout* and column vector ***x***. However, the notation implies only a formal convention, which does not mean computational expression, and thus, it is possible to consider that:

However:

Note, the derivatives follows numerator layout so that gradient vector ∇*f*(***x***) and gradient matrix ∇*f*(***X***) are 1x*n* row vector and *n*x*m* matrix when ***x*** and ***X*** are *n*x1 column vector and *m*x*n* matrix.

Where the superscript “*T*” denotes transposition operator of vector and matrix. Anyhow the notations *f*’(.) and *d*(.)/*d*(.) are most common notations for denoting derivative.

Given ***y*** = (*y*1, *y*2,…, *yp*)*T* is *p*x1 column vector, function ***y*** = ***f***(*x*) whose *x* is scalar and ***y*** is vector, function ***y*** = ***f***(***x***) whose ***x*** and ***y*** are vectors, and function ***y*** = ***f***(***X***) whose ***X*** is matrix and ***y*** is vector are called *vector-by-scalar* function, *vector-by-vector* function, and *vector-by-matrix* function, respectively whose derivatives are called *tangent vector* ***f***’(*x*), *Jacobian matrix* *J****f***(***x***), and 3-dimesion *tangent tensor* ***f***’(***X***).

Note, thought:

But really:

Due to numerator layout and column vector ***x***. However, the notation implies only a formal convention, which does not mean computational expression, and thus, it is possible to consider that:

However:

Note, the derivatives follows numerator layout so that tangent vector ***f***’(*x*), Jacobian matrix *J****f***(***x***), and tangent tensor ***f***’(***X***) are *p*x1 column vector, *p*x*n* matrix, and 3-dimension *tensor*. Each element of tangent tensor ***f***’(***X***) is a *n*x*m* gradient matrix ∇*yi*(***X***):

Given ***Y*** is *p*x*q* matrix variable:

Function ***Y*** = ***F***(*x*) whose *x* is scalar and ***Y*** is vector, function ***Y*** = ***F***(***x***) whose ***x*** is vector and ***Y*** is matrix, and function ***Y*** = ***F***(***X***) whose ***X*** and ***Y*** are matrices and ***y*** are called *matrix-by-scalar* function, *matrix-by-vector* function, and *matrix-by-matrix* function, respectively whose derivatives are called *tangent matrix* ***F***’(*x*), 3-dimension *Jacobian tensor* *J****F***(***x***), and 4-dimesion Jacobian tensor *J****F***(***X***).

Note, though:

But really:

Due to numerator layout and column vector ***x***. However, the notation implies only a formal convention, which does not mean computational expression, and thus, it is possible to consider that:

However:

Each element of Jacobian tensor *J****F***(***x***) is a 1x*n* gradient vector ∇*yij*(***x***):

Each element of Jacobian tensor *J****F***(***X***) is a *n*x*m* gradient matrix ∇*yij*(***X***).

Because matrix-by-matrix function ***Y*** = ***F***(***X***) is the most general case, how to determine its derivatives, particularly, Jacobian tensor *J****F***(***X***) is focused here with note that there are many equations in literature for calculating multidimensional derivatives of vector functions. Because determining 4-dimension tensor *J****F***(***X***) is hazard, there are two methods of solving the problem of derivative tensor such as *vectorization method* and *differential method* (Gemini 2025). Firstly, vectorization method is mentioned that given matrix ***X***, vectorization operator on ***X*** denoted vec(***X***) is the concatenation columns of ***X*** into single *mn*x1 column vector, for instance:

Note,

For instance, given matrix *A*:

Its *vectorized vector* vec(*A*) is:

Given three matrices *P*, *Q*, and *R*, following is an important property of vectorization:

|  |  |
| --- | --- |
|  | (A1.1) |

Where the binary operator denotes *Kronecker product*. Given two matrices *A* and *B* whose dimensions are *m*x*n* and *p*x*q*.

Their Kronecker product is (Wikipedia 2025):

Dimension of matrix *AB* is *mp* x *nq*. For instance, if

Then the Kronecker product *A*B is 6x4 matrix:

Let ***x*** = vec(***X***) and ***y*** = vec(***Y***) whose dimensions are 1x*mn* and 1x*pq*, respectively, matrix-by-matrix function ***Y*** = ***F***(***X***) corresponds with vector-by-vector function ***y*** = vec(***Y***) = ***F***(vec(***X***)) = ***F***(***x***) and Jacobian tensor *J****F***(***X***) corresponds with Jacobian matrix *J****f***(***x***).

In general, it is possible to identify *J****F***(***X***) with *J****F***(vec(***X***))*d*(vec(***X***)):

|  |  |
| --- | --- |
|  | A1.2 |

Although *J****F***(***X***) is 4-dimesion tensor and *J****F***(vec(***X***)) is *pq* x *mn* matrix because ***X*** and ***Y*** are *m*x*n* matrix and *p*x*q* matrix, respectively, they are mutually equivalent, possibly denoted *J****F***.

Of course, it is easy to transform vectors vec(***X***) and vec(***Y***) backward original matrices ***X*** and ***Y***, respectively.

Note here, ***x*** = vec(***X***) and ***y*** = vec(***Y***) are called *vectorized vectors* and ***F***(vec(***X***)) is called *vectorized function* whereas *J****F***(vec(***X***)) is called *vectorized Jacobian matrix*.

Given two matrix-by-matrix functions ***Y*** = ***F***(***X***) and ***Z*** = ***G***(***Y***), chain rule for vectorized function which is important to vectorization method is specified as follows:

|  |  |
| --- | --- |
|  | (A1.3) |

Which is the same to:

Note,

Please pay attention that:

|  |  |
| --- | --- |
|  | (A1.4) |

Secondly, differential method is mentioned here with note that concept of *differential* is always associated with concept of derivative. Let *d****Y*** be the differential of ***Y***, we have:

Where,

Note, *d****X*** is differential of ***X***, which means that the derivative ***F***’(***X***) is the *linear map* that transforms differential of ***X*** into differential of ***Y***. It may be easier or more possible to determine directly the differential *d****Y*** for determining indirectly the derivative ***F***’(***X***) due to two following reasons. The first reason is that there is an equivalence between differential *d****Y*** and derivative ***F***’(***X***) as follows:

The second reason is that there are some handy properties related to differential *d****Y***. For instance, given ***X*** and ***Y*** are matrix variables whereas *A*, *B*, *C*, and *D* are matrix constants, following are some important rules related to matrix differential.

Moreover, some functions are appropriate to context of differential, for instance, given scalar-by-matrix function *y* = *f*(***X***) then its differential *df*(***X***) is *trace* of product of its derivative and differential *d****X***:

|  |  |
| --- | --- |
|  | (A1.5) |

The equation above specifies the fundamental relationship between scalar function *f*(***X***) and its gradient matrix ∇*f*(***X***) via trace operator with note that tr(.) denotes trace operator which is the sum of all diagonal elements of a squared matrix. Note, (∇*f*(***X***))*Td****X*** is always squared matrix.

Given squared matrix ***Y***, we always have:

|  |  |
| --- | --- |
|  | (A1.6) |

In general, taking derivative of matrix-by-matrix function ***Y*** = ***F***(***X***) includes two steps: 1) taking differential *d****Y*** so that *d****Y*** is linear map of *d****X*** and 2) if *d****X*** can be isolated from the linear map when it is at the end of linear map like *d****Y*** = *A*1*A*2…*Akd****X*** then the derivative ***F***’(***X***) is *A*1*A*2…*Ak*, otherwise, *d****Y*** is vectorized so as to isolate vec(*d****X***) from vec(*d****Y***) like vec(*d****Y***) = *B*1*B*2…*Bl*vec(*d****X***) then the derivative ***F***’(***X***) is ***f***’(vec(***X***)) = *B*1*B*2…*Bl*. The isolating vec(*d****X***) from ***F***’(***X***) is often applied by the known property of vectorization given three matrices *P*, *Q*, *R*.

Where the binary operator denotes Kronecker product. Particularly, for taking derivative of scalar-by-matrix function *y* = *f*(***X***), the following property is often applied into isolating *d****X*** from the concerned derivative *f*’(***X***).

Note in the case that ***X*** is simplified as column vector ***x*** as ***X*** = ***x*** then ∇*f*(***x***) is transposed again to be row vector.

It is possible to make vectorization for taking derivative of scalar-by-matrix function with note that trace operator has a sequential property to push the *d****X*** to the end of *df*(***X***) for isolation of *d****X***. Without loss of generality, given three matrices *A*, *B*, and *C*, the *sequential property* of trace operator is specified as follows:

|  |  |
| --- | --- |
|  | (A1.7) |

Anyhow, *isolation of differential* *d****X*** is always concerned for taking derivative.

For example, given ***F***(***X***) = *A****X****B* + *C* where *A*, ***X***, *B*, and *C* are appropriate matrices in this product in which ***X*** is variable, we will calculate its derivative ***F***’(***X***). Indeed, its differential is:

|  |  |
| --- | --- |
|  | (A1.8) |

It is necessary to isolate *d****X*** from ***F***’(***X***) by moving it to the end of *d****F*** and thus, *d****F*** is vectorized as follows:

Applying the following property given three matrices *P*, *Q*, *R*:

We have:

As a result, derivative of ***F***(***X***) = *A****X****B* + *C* which is Jacobian matrix of ***F***(***X***) is determined as follows:

Another example is that given scalar-by-matrix function *y* = *f*(***X***) whose formulation is unknown yet (we will research optimal function which is a popular scalar-by-matrix function later), but its differential is

Indeed, we have:

As a result, we obtain the derivative *f*’(***X***) = (*BA*)*T*.

Another example is that given two following matrix-by-matrix functions ***F***(.) and ***G***(.), taking derivatives of ***G*** with regard to ***X*** and ***Y***.

Where ***X***, ***Y***, ***Z***, ***W***, and ***K*** are matrix variables while *A*, *B*, *C*, *D*, *P*, *Q*, *R*, and *S* are matrix constants. Let,

So that

Let,

The ultimate purpose here is to determine the derivatives of ***G*** with regard to ***X*** and ***Y*** which are Jacobian tensors (matrices) *J****G***(***X***) and *J****G***(***Y***) related to differentials ∂***G***(***X***) and ∂***G***(***Y***). Vectorising differential ∂***G***(***X***), we obtain:

By applying the property , we have:

Where denotes Kronecker product. Due to:

Which means:

By applying the property , we obtain:

Therefore,

Which implies:

Therefore, derivative of ***G*** with regard to ***X*** is:

Where,

Therefore, derivative of ***G*** with regard to ***Y*** is:

Another example is to take derivative of scalar-by-matrix function *l*(***K***) called likelihood function which is defined here as the negative of loss function which in turn is squared *Frobenius norm* of the difference between two matrices ***K*** and ***K***’ where ***K***’ is considered as real output value from environment (not from ***K*** = ***G***(***Z***) = *P****Z****Q* + *R****W****S*)). Following is the *loss function* as Frobenius norm:

Therefore, likelihood function *l*(***K***) is defined as follows:

So that

Of course, we always have:

The ultimate purpose of this example is to find out the maximizers ***Z***\* and ***X****\** so that likelihood *l*(*.*) gets maximal at ***Z***\* and ***X***\*, which is the optimization problem.

According to stochastic gradient descent (SGD) algorithm, ***Z***\* and ***X****\** are estimated iteratively as follows:

Where *γ* (0 < *γ* ≤ 1) is learning rate. Consequently, the optimization problem focuses now on determining the two gradient matrices ∇*l*(***Z***) and ∇*l*(***X***) which are derivatives of likelihood with regard to ***Z*** and ***X***, respectively. Note,

Differential *dl* of likelihood with regard to ***Z*** within vectorization is:

Due to:

Which implies:

Therefore, gradient ∇*l*(***Z***) is totally equivalent to gradient ∇*l*(vec(***Z***)).

Similarly, differential *dl* of likelihood with regard to ***X*** within vectorization is:

Due to:

Which implies:

Therefore, gradient ∇*l*(***Z***) is totally equivalent to gradient ∇*l*(vec(***Z***)).

As a result, ***Z***\* and ***X****\** are estimated iteratively according to SGD as follows:

|  |  |
| --- | --- |
|  | (A1.9) |

|  |  |
| --- | --- |
|  | (A1.10) |

|  |  |
| --- | --- |
|  | (A1.11) |

Where,

The vectorization is reversed so as to obtain ***Z*** and ***X*** as ***Z*** = vec–1(***Z***) and ***X*** = vec–1(***X***). Please pay attention that ∇*l*(vec(***Z***)) is column vector here for easy computation although it is row vector according to numerator layout convention because it is gradient of scalar-by-matrix function *l*(vec(***Z***).

**A2. Convolutional neural network**

Although artificial neural network (ANN) produces amazing results in machine learning including supervised learning, unsupervised learning, and reinforcement learning, its serious drawback is that there is a huge memory capacity for storing parametric weights, especially in the case of large numbers of neurons, which make ANN implementation in memory impossible. However, in the case of 2-dimension data like image, memory capacity is reduced significantly by a so-called *convolution neural network* (*CNN*). For instance, given 100x100 image containing 10000 pixels requires (100x100)2 bytes at least for storing one parametric weight with only 2-layer ANN with suppose 1 byte for a number from 0 to 255. Fortunately, when 100x100 image is considered as a so-called *convolutional layer* which is filtered by a 2x2 filter, such CNN layer is reduced into 10000/4 = 2500 pixels. As a result, a CNN has two parts: 1) the first part is a sequential set of convolutional layer layers which of them is filtered by a *k*x*k* filter, and 2) the *fully connected network* called also *dense network* is a usual feedforward network (FFN). The last layer of the dense network specifies particular image processing applications such as image classification and image recognition. Until now (2025) CNN is the most popular approach to image processing domain. Anyhow, the most important aspect of CNN is its convolutional layers. Given an image as 2-dimension *m*x*n* matrix denoted ***X*** = (*xij*)*m*x*n*, a *filter* also called *kernel* is a *k*x*k* square matric denoted *W* = (*wij*)*k*x*k*, which is often 3x3 or 5x5 matrix. Given pixel ***X***[*i*][*j*] = *aij* = *ai*,*j* at the *i*th row and *j*th column of matrix ***X***, the respective value *y* which is a cell of convolutional layer ***Y***, which is called *convolutional value* or convolutional cell, is calculated by a so-called *filter operator* which is the sum of multiplications of neighbor pixels and *W* = (*wij*)*k*x*k*.

Where *θ* is the *bias* which can be ignored in theoretical study such that:

Activation function *f*(.) can be applied into filter operator like ANN does:

|  |  |
| --- | --- |
|  | (A2.1) |

But the common activation function for CNN is Rectified Linear Unit (ReLU) which limits its input in the interval [*a*, *b*].

If the interval [*a*, *b*] is [0, 1] then:

Because ReLU does not change its input and its derivative is 1, it can be ignored in theoretical study such that:

Indeed, the filter *W* is slid over entire image ***X*** so as to complete convolutional layer ***Y*** and so, the movement of filter is jumped step-by-step where the step length is called *stride* denoted *s*. As usual, stride *s* is in the range [1, *k*] such that 1 ≤ *s* ≤ *k* where *k* is often called *filter size* for filter *W* = (*wij*)*k*x*k*. If image ***X*** is *n*x*n* matrix and filter *W* is *k*x*k* matrix given stride *s*, then the size *p* of convolution (*C*)*p*x*p* is:

As a convention, we can denote:

|  |  |
| --- | --- |
|  | (A2.2) |

If filter *W* is aligned at the center of the pixel that it is applied, the filter operator is modified as follows:

When filter *W* is aligned at the center of the pixel that is applied, the filtered size of convolutional layer should be subtracted by the half size of kernel although its entire size *p* is not changed, which means:

As a convention, we can denote:

Because the entire size p is kept intact, there are *k*/2 rows and *k*/2 columns which are padded by zero values, which is called *zero-padding* technique.

The convolutional layer has two aspects: 1) significant reducing the size of dense network, and 2) extracting features of image. For the first aspect, given *n*x*n* image, suppose the stride is kernel size as *s* = *k*, if there are *w* convolutional layers, then the input size of dense network is (*n*\**n*) / ((*k*\**k*)*w*) neurons instead of standing for *n*\**n* neurons. For the second aspect, the features are based on how to specified filters for what purposes are, for example, blurring, edge detection.

Training convolutional neural network (CNN) is simpler than training traditional artificial neural network (ANN) and so, *stochastic gradient descent* (*SGD*) algorithm is applied again into training CNN which is to learn parametric filter and parametric bias. For instance, given three sequential convolutional layers ***X***, ***Y***, and ***Z*** whose parametric filters are *W*(***Y***) and *W*(***Z***) and whose parametric biases are *θ*(***Y***) and *θ*(***Z***), respectively, their filter operators are:

Training CNN is to learn parameters *W*(***Y***), *θ*(***Y***), *W*(***Z***), and *θ*(***Z***) where ***X*** is input layer and ***Z*** is output layer in CNN. Let *l*(*zij*) be the *likelihood function* which takes *zij* as its input, as usual, *l*(*zij*) is the negative of *error function*:

Later on, we will know that *l*(*zij*) is propagated backward from dense network connected directly to CNN. Let *l*(***Z***) be the entire likelihood function which is the mean of *l*(*zij*) over all *zij* (s) belonging to ***Z***.

Where the notation |***Z***| denotes the size of layer ***Z***, for instance, if ***Z*** is *m*x*n* matrix, then its size |***Z***| is *m*\**n*. The parameters *W*(***Z***) and *θ*(***Z***) of the last layer ***Z*** are estimated firstly:

According SGD, parametric filter *wuv*(***Z***) and parametric bias *θ*(***Z***) are estimated iteratively by addition of itself and gradients of likelihood function *l*(***Z***) with respect to *wuv*(***Z***) and *θ*(***Z***), respectively, increased by step of learning rate *γ*. As a result, training CNN is essentially to calculate these gradients. For layer ***Z***, we have:

Where,

Let and be gradients of likelihood function *l*(*zij*) with respect to *wuv*(***Z***) and *θ*(***Z***), respectively, as follows (Gemini 2025):

We obtain the equation to learn parametric filter *wuv*(***Z***) and parametric bias *θ*(***Z***) at output layer ***Z***.

|  |  |
| --- | --- |
|  | (A2.3) |

The quantity is also called elemental parametric error at layer ***Z***. The error will be propagated backward from layer ***Z*** back to layer ***Y*** and so, let be the error of every element at layer ***Y*** (Gemini 2025):

In practice, the error of every element *yij* at layer ***Y*** is the mean of over kernel *W*(***Z***) as follows:

Which implies:

|  |  |
| --- | --- |
|  | (A2.4) |

Where the notation |*W*(***Z***)| denotes the size of filter *W*(***Z***), for instance, if *W*(***Z***) is *k*x*k* matrix, then its size |*W*(***Z***)| is *k*\**k*.

The parameters *W*(***Y***) and *θ*(***Y***) of the middle layer ***Y*** are estimated secondly:

So that they can be estimated according to SGD:

Where,

Therefore, estimating parameters *W*(***Y***) and *θ*(***Y***) is to determine their gradients and . We have:

We obtain the equation to learn parametric filter *wuv*(***Z***) and parametric bias *θ*(***Z***) at output layer ***Z***.

|  |  |
| --- | --- |
|  | (A2.5) |

The quantity is also called elemental parametric error at layer ***Y*** whereas the quantity is the error of every element *yij* at layer ***Y*** which will be continuously propagated. As a result, parametric filters and parametric biases of entire CNN are estimated by association of stochastic gradient descent (SGD) algorithm and backpropagation algorithm as follows:

In general case of K-layer CNN whose layers are ***X***0, ***X***1, ***X***2,…, ***X****K*, then training CNN within association of SGD and backpropagation algorithm is summarized from *k*=1 to *k*=*K* as follows:

|  |  |
| --- | --- |
|  | (A2.6) |

So that CNN parameters are estimated iteratively and backward according to SGD:

Note, is the element of layer ***X****k* at the *i*th row and *j*th column. The auxiliary problem of training CNN is how to calculate the *last convolutional error* at the output layer ***X****K*. Fortunately, although convolutional last layer ***X****K* is not compared with any output data in CNN because the context of convolutional layer is to reduce image size and to extract features of image, the last convolutional error is propagated backward from the core last bias of dense network (fully connected network) if ***X****K* connects directly to dense network. Let denote the last convolutional error (matrix) including all at the last convolutional layer ***X****K*. Suppose the *N*-layer dense network has *N* layers such as ***Y***0, ***Y***1, ***Y***2,…, ***Y****N*, the last convolutional error is translated as the bias of the dense network at its first layer ***Y***0, propagated from the *core last bias* as follows:

|  |  |
| --- | --- |
|  | (A2.7) |

Where vec(.) represents vectorization technique which makes a matrix flatten as column vector whereas *Un* and *Vn* are parametric weight matrices of layer ***Y****n* while *g*(.) is activation function of dense network and the notation denotes Kronecker product.

As usual, the core last bias is derivative of the squared error of real output ***Y***’*N* and computed output ***Y****N*.

Please pay attention that the vectorized function vec(.) is invertible, for instance, ***Y****N* = vec–1 (vec(***Y****N*)). In practice, layers of dense network are flattened vectors such as ***y***0, ***y***1, ***y***2,…, ***y****N*, the last convolutional error becomes simpler:

|  |  |
| --- | --- |
|  | (A2.8) |

Where *Wn* is parametric weight matrix of layer ***y****n* such that:

Where ***y***’*N* is the real output of dense network from environment. In general, that the convolutional error is propagated backward from the core last bias of dense network implies that the likelihood function is propagated backward from dense network to convolutional network too.

# References

Chandrakant, K. (2023, March 24). *Reinforcement Learning with Neural Network*. (Baeldung) Retrieved from Baeldung website: https://www.baeldung.com/cs/reinforcement-learning-neural-network

Choudhary, A. (2019, April 18). *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python*. Retrieved from Analytics Vidhya website: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python

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.

Nguyen, L. (2022). *Mathematical Approaches to User Modeling* (1st ed.). (O. Sabazova, Ed.) Moldova: Eliva Press. Retrieved February 16, 2022, from https://www.elivapress.com/en/book/book-6035512576

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. (2002, October 22). *Dynamic programming*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Dynamic\_programming

Wikipedia. (2002, July 31). *Reinforcement learning*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Reinforcement\_learning

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. (2004, November 2). *Markov decision process*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Markov\_decision\_process

Wikipedia. (2004, December 15). *Q-learning*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Q-learning

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](#_Toc204773554)

[1. Introduction 1](#_Toc204773555)

[2. More about learning algorithm 21](#_Toc204773556)

[3. Convergence of learning algorithm 38](#_Toc204773557)

[4. Recurrent network 44](#_Toc204773558)

[5. Self-organizing network 49](#_Toc204773559)

[6. Reinforcement learning 52](#_Toc204773560)

[7. Conclusions 64](#_Toc204773561)

[Appendices 65](#_Toc204773562)

[References 80](#_Toc204773563)