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*.

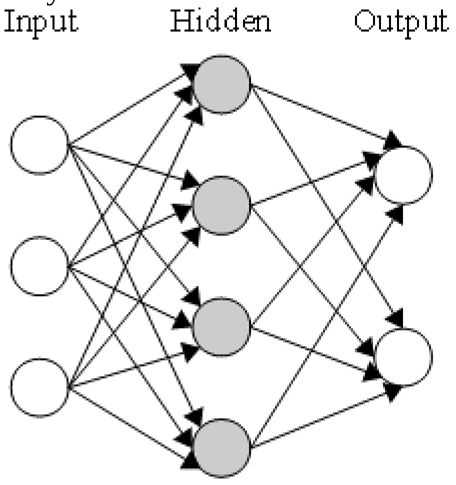

Figure 1.1. Simpler topology of NN with three layers such as input layer, hidden layer, and output layer

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 x_i and y_i 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 y_i 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 w_{ij} 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.

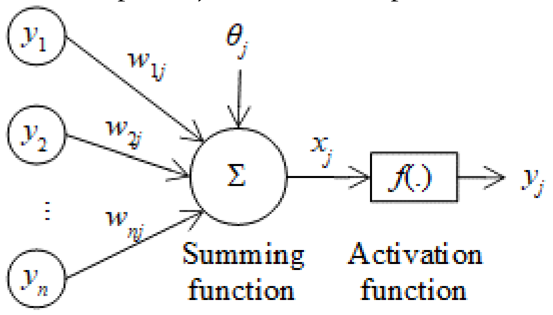

Figure 1.2. Process of computing output of a unit

For example, as seen in figure 1.2, given a concerned unit *k*, suppose there are previous units whose outputs y_j (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.

$$\begin{aligned} x_k &= \sum_j w_{jk}y_j + \theta_k \\ y_k &= f_k(x_k) \end{aligned} \tag{1.1}$$

Or shortly:

$$y_k = f_k\left(\sum_i w_{jk}y_j + \theta_k\right)$$

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

$$x_k = \sum_j w_{jk} y_j + \theta_k$$

$$y_k = f(x_k)$$

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

$$y_k = f\left(x_k = \sum_j w_{jk} y_j + \theta_k\right) \tag{1.2}$$

The parameters of propagation rule are weights w_{jk} and biases θ_k in which weights are most important. Conversely, it is possible to consider propagation rule as function of variables w_{jk} and θ_k. 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:

$$y_k(t+1) = f\left(x_k(t+1) = \sum_j w_{jk} y_j(t) + \theta_k\right) \tag{1.3}$$

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, x_i and y_i), hidden units in hidden layer are indexed by h (for instance, x_h and y_h), and output units in output layer are indexed by o (for instance, x_o and y_o). 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 $x_i = y_i$ and $\theta_i = 0$. 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 x_k and output y_k. Moreover, let v_k be the actual value of unit k taken from experts, environment, database, states, etc. The actual value v_k can be equal to or different from the output v_k with note that v_k is derived from propagation rule. The actual value v_k 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 s_k. By default, all units are clamped and so, the *clamped input* s_k is the same to the input x_k as $s_k = x_k$ by default.
- A set of units j connects to it. Each connection is quantified by a weight w_{jk}.
- A bias value θ_k 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 y_k 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, $v_k^{(1)}$, $v_k^{(2)}$,…, and hence, given a *pattern p* (Kröse & Smagt, 1996, p. 19) there is a desired output $v_k^{(p)}$ corresponding to pattern p. For easily understandable explanation, if $v_k^{(p)}$ is taken from a database table, p indicates the pth row in the table. As a convention, let $x_k^{(p)}$, $y_k^{(p)}$, $v_k^{(p)}$, and $s_k^{(p)}$ be input, output, desired output, clamped input of unit k within the p pattern, respectively or they can be called the pth input, output, desired output, and clamped input of unit k, respectively. With pattern p, propagation rule is rewritten exactly as follows:

$$\begin{gathered} s_k^{(p)} = \sum_{j \in N(k)} w_{jk} y_j^{(p)} + \theta_k \\ y_k^{(p)} = f\left(s_k^{(p)}\right) \end{gathered} \tag{1.4}$$

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:

$$s_k^{(p)}(t+1) = \sum_{j \in N(k)} w_{jk} y_j^{(p)}(t) + \theta_k$$

$$y_k^{(p)}(t+1) = f\left(s_k^{(p)}(t+1)\right)$$

Propagation rule essentially transforms inputs to outputs but an output y_k may not totally equal to desired output v_k when it is often approximated to v_k. 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 $v_k = v(s_k)$ (= $v(x_k)$). 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 $x_1, x_2,…, x_k,…$ is denoted as $\boldsymbol{x} = (x_1, x_2,…, x_k,…)^T$ which is called *input vector* where the superscript "T" denotes transposition operator of vector and matrix. The set of outputs y_1, $y_2,…, y_k,…$ is denoted as $\boldsymbol{y} = (y_1, y_2,…, y_k,…)^T$ which is called *output vector*. The set of desired outputs $v_1, v_2,…, v_k,…$ is denoted as $\boldsymbol{v} = (v_1, v_2,…, v_k,…)^T$ which is called *desired output vector*. The set of clamped inputs $s_1, s,…, s_k,…$ is denoted as $\boldsymbol{s} = (s_1, s_2,…, s_k,…)^T$ which is called *clamped input vector*. Input vector, output vector, desired vector, and clamped input vector with p pattern are denoted $\boldsymbol{x}^{(p)}$, $\boldsymbol{y}^{(p)}$, $\boldsymbol{v}^{(p)}$, and $\boldsymbol{s}^{(p)}$, respectively. The set of input vector over entire input layer and desired output vector over entire output layer composes a sample or training dataset $D = \{\boldsymbol{x}^{(p)}, \boldsymbol{v}^{(p)}\}$ for learning NN where p = 1, 2, 3, etc. By default, all units are clamped in NN and so we have $D = \{\boldsymbol{x}^{(p)}, \boldsymbol{v}^{(p)}\} = \{\boldsymbol{s}^{(p)}, \boldsymbol{v}^{(p)}\}$ 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:

$$f(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

- *Piecewise-linear function* takes on values according to amplification factor in a certain region of linear operation. The formula of piecewise-linear function is:

$$f(x) = \begin{cases} 0 \text{ if } x \leq -\frac{1}{2} \\ x \text{ if } -\frac{1}{2} \leq x \leq \frac{1}{2} \\ 1 \text{ if } \frac{1}{2} \leq x \end{cases}$$

- *Sigmoid function* or logistic function takes on values in range [0, 1] or [–1, 1]. A popular formula of sigmoid function is:

$$f(x) = \frac{1}{1+e^{-x}} \tag{1.5}$$

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 w_{jk} from unit j to unit k, a new version of w_{jk} after learning process at time point t is updated by weight deviation Δw_{jk} as follows:

$$w_{jk}(t+1) = w_{jk}(t) + \Delta w_{jk}$$

Or shortly:

$$w_{jk} = w_{jk} + \Delta w_{jk} \tag{1.6}$$

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

$$\theta_k(t+1) = \theta_k(t) + \Delta\theta_k$$

Or shortly:

$$\theta_k = \theta_k + \Delta\theta_k \tag{1.7}$$

The equation above is called *bias update rule* and hence, bias update rule focuses on how to calculate bias deviation $\Delta\theta_k$ 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 Δw_{jk} and $\Delta\theta_k$ 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 Δw_{jk} (also w_{jk}) is proportional to product of output of unit j and output of unit k as follows (Kröse & Smagt, 1996, p. 18):

$$\Delta w_{jk} = \gamma y_j y_k \tag{1.8}$$

Where the positive constant γ which is called learning rate ($0 < \gamma \leq 1$) specifies power of the proportionality, which relates to speed of learning process. In simplest case, it is 1 as $\gamma = 1$. Both y_j and y_k are results of propagation rule. Let v_k be desired output of unit k from environment or database, delta rule indicates that Δw_{jk} (also w_{jk}) is proportional to product of output value of unit j and output deviation of unit k as follows (Kröse & Smagt, 1996, p. 18):

$$\Delta w_{jk} = \gamma y_j (v_k - y_k) \tag{1.9}$$

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).

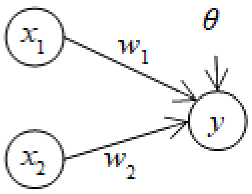

Figure 1.3. Perceptron topology

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):

$$x = w_1x_1 + w_1x_1 + \theta$$
$$y = f(x) = \begin{cases} 1 \text{ if } x > 0 \\ -1 \text{ otherwise} \end{cases} \tag{1.10}$$

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 $w_1x_1 + w_2x_2 + \theta = 0$. Weight update rule of Perceptron is:

$$w_i = w_i + \Delta w_i, \forall i = \overline{1,2}$$

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

$$\Delta w_i = \begin{cases} x_i v \text{ if } y \neq v \\ 0 \text{ if } y = v \end{cases}, \forall i = \overline{1,2} \tag{1.11}$$

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

$$\theta = \theta + \Delta\theta$$

Perceptron learning algorithm calculates bias deviation $\Delta\theta_i$ as follows (Kröse & Smagt, 1996, p. 25):

$$\Delta\theta = \begin{cases} v \text{ if } y \neq v \\ 0 \text{ if } y = v \end{cases} \tag{1.12}$$

For example, with initialized values $w_1 = 1$, $w_2 = 1$, and $\theta = 0$, given sample $x_1 = 1$, $x_2 = 2$, and $v = 1$, Perceptron weights and biases are updated as follows:

$$\begin{aligned} x &= w_1x_1 + w_1x_1 + \theta = 3 \\ y &= 1 \text{ due to } x > 0 \\ \Delta w_1 &= 0 \text{ due to } y = v = 1 \\ \Delta w_2 &= 0 \text{ due to } y = v = 1 \\ \Delta\theta &= 0 \text{ due to } y = v = 1 \\ w_1 &= w_1 + \Delta w_1 = 1 \\ w_2 &= w_2 + \Delta w_2 = 1 \\ \theta &= \theta + \Delta\theta = 0 \end{aligned}$$

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 x_i (s). Propagation rule of Adaline is (Kröse & Smagt, 1996, p. 28):

$$y = \sum_i w_i x_i + \theta \tag{1.13}$$

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

$$\varepsilon(w_i,\theta) = \sum_p \varepsilon^{(p)}(w_i,\theta) \tag{1.14}$$

Where (Kröse & Smagt, 1996, p. 28),

$$\varepsilon^{(p)}(w_i,\theta) = \frac{1}{2}\left(v^{(p)} - y^{(p)}\right)^2 = \frac{1}{2}\left(v^{(p)} - \left(\sum_i w_i x_i^{(p)} + \theta\right)\right)^2 \tag{1.15}$$

Note, $\varepsilon^{(p)}(w_i, \theta)$, which is function of w_i and θ, is the squared error at pattern p or the pth squared error in short. According to least squares method, the optimal $(w_i^{**}, \theta^{**})^T$ is minimizer of the total error.

$$(w_i^{**},\theta^{**}) = \underset{(w_i,\theta)}{\operatorname{argmin}}\, \varepsilon(w_i,\theta)$$

By feeding successively each {$x^{(p)}$, $v^{(p)}$} or summing all squared errors $\varepsilon^{(p)}(w_i, \theta)$, it is possible to calculate a minimizer (w_i^*, θ^*) at each pattern p, which minimizes the pth squared error $\varepsilon^{(p)}(w_i, \theta)$.

$$(w_i^{*},\theta^{*}) = \underset{(w_i,\theta)}{\operatorname{argmin}}\, \varepsilon^{(p)}(w_i,\theta) \tag{1.16}$$

After feeding all patterns one by one, the final minimizer $(w_i^*, \theta^*)^T$ is expected to minimize the total squared error $\varepsilon(w_i, \theta)$ like $(w_i^{**}, \theta_i^{**})$. Stochastic gradient descent (SGD) method is used to search for the maximizer $(w_i^*, \theta^*)^T$ with the target function $\varepsilon^{(p)}(w_i, \theta)$. 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 $\varepsilon^{(p)}(w_i, \theta)$.

$$\begin{gathered}(w_i,\theta)^{(p)} = (w_i,\theta)^{(p)} - \gamma\nabla\varepsilon^{(p)}(w_i,\theta) \\ \nabla\varepsilon^{(p)}(w_i,\theta) = \left(\frac{\partial\varepsilon^{(p)}(w_i,\theta)}{\partial w_i}, \frac{\partial\varepsilon^{(p)}(w_i,\theta)}{\partial\theta}\right)\end{gathered} \tag{1.17}$$

Note, the gradient of $\varepsilon^{(p)}(w_i, \theta)$ denoted $\nabla\varepsilon^{(p)}(w_i, \theta)$ is row vector of partial derivatives of $\varepsilon^{(p)}(w_i, \theta)$ (Kröse & Smagt, 1996, p. 28). Due to (Kröse & Smagt, 1996, pp. 28-29):

$$\frac{\partial\varepsilon^{(p)}(w_i,\theta)}{\partial w_i} = -x_i^{(p)}\left(v^{(p)} - y^{(p)}\right)$$
$$\frac{\partial\varepsilon^{(p)}(w_i,\theta)}{\partial\theta} = -\left(v^{(p)} - y^{(p)}\right)$$

We have:

$$\nabla\varepsilon^{(p)}(w_i,\theta) = -\left(x_i^{(p)}\left(v^{(p)} - y^{(p)}\right), v^{(p)} - y^{(p)}\right)$$

As a result, weight deviation and bias deviation are determined based on γ and the gradient of $\varepsilon^{(p)}(w_i, \theta)$ as follows (Kröse & Smagt, 1996, p. 29):

$$\begin{gathered}\Delta w_i^{(p)} = -\gamma\frac{\partial\varepsilon^{(p)}(w_i,\theta)}{\partial w_i} = \gamma x_i^{(p)}\left(v^{(p)} - y^{(p)}\right) \\ \Delta\theta^{(p)} = -\gamma\frac{\partial\varepsilon^{(p)}(w_i,\theta)}{\partial\theta} = \gamma\left(v^{(p)} - y^{(p)}\right)\end{gathered} \tag{1.18}$$

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 $\varepsilon^{(p)}(w_i, \theta)$ at pattern p and so, it is easy to extend such equation for the total squared error function $\varepsilon(w_i, \theta) = \sum_p \varepsilon^{(p)}(w_i, \theta)$ over all patterns:

$$\Delta w_i = \sum_p \Delta w_i^{(p)} = \sum_p \gamma x_i^{(p)}\left(v^{(p)} - y^{(p)}\right)$$

$$\Delta\theta = \sum_p \Delta\theta^{(p)} = \sum_p \gamma\left(v^{(p)} - y^{(p)}\right)$$

The extension is easy to be asserted because the squared error function $\varepsilon^{(p)}(w_i, \theta)$ and the total squared error function $\varepsilon(w_i, \theta)$ 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:

$$\begin{gathered} w_i = w_i + \Delta w_i \\ \theta = \theta + \Delta\theta \end{gathered} \tag{1.19}$$

Where,

$$y = \sum_i w_i x_i + \theta$$

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:

$$x_k = \sum_j w_{jk} y_j + \theta_k$$

$$y_k = f(x_k)$$

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 v_o, the squared error function of output unit o for normal NN is (Kröse & Smagt, 1996, p. 34):

$$\varepsilon(y_o) = \varepsilon(w_{ho}, \theta_o) = \frac{1}{2}(v_o - y_o)^2 \tag{1.20}$$

Where,

$$y_o = f\left(x_o = \sum_h w_{ho} y_h + \theta_o\right)$$

Note that all previous outputs y_h were determined. Moreover, by default, all units are clamped and so, the clamped input s_o is the same to the input x_o as $s_o = x_o$ 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.

$$\varepsilon(y_o) = \sum_p \varepsilon^{(p)}(y_o) = \sum_p \frac{1}{2}\left(v_o^{(p)} - y_o^{(p)}\right)^2$$

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

$$\varepsilon(y_o) = \varepsilon(w_{ho}, \theta_o) = \varepsilon^{(p)}(y_o) = \frac{1}{2}\left(v_o^{(p)} - y_o^{(p)}\right)^2 = \frac{1}{2}(v_o - y_o)^2$$

Recall that weight deviation Δw_{ho} and bias deviation $\Delta\theta_o$ are determined based on the gradient of the squared error function $\varepsilon(y_o)$ according to stochastic gradient descent (SGD) method for minimizing the squared error function $\varepsilon(y_o)$.

$$(w_{ho}, \theta_o) = (w_{ho}, \theta_o) - \gamma\nabla\varepsilon(w_{ho}, \theta_o)$$

Note, the gradient of $\varepsilon(y_o)$ with regard to w_{ho} and θ_o is row vector of partial derivatives of $\varepsilon(y_o)$ with regard to w_{ho} and θ_o as follows:

$$\nabla\varepsilon(y_0) = \nabla\varepsilon(w_{ho}, \theta_o) = \left(\frac{\partial\varepsilon(y_0)}{\partial w_{ho}}, \frac{\partial\varepsilon(y_0)}{\partial\theta_o}\right)$$

By SGD, weight deviation Δw_{ho} and bias deviation $\Delta\theta_o$ are products of learning rate and descending direction of $\varepsilon(y_o)$ which is the opposite of the gradient $\nabla\varepsilon(w_{ho}, \theta_o)$.

$$\Delta w_{ho} = -\gamma\frac{\partial\varepsilon(y_o)}{\partial w_{ho}}$$
$$\Delta\theta_o = -\gamma\frac{\partial\varepsilon(y_o)}{\partial\theta_o}$$

Due to chain rule in derivation:

$$\frac{\partial\varepsilon(y_0)}{\partial w_{ho}} = \frac{\partial\varepsilon(y_0)}{\partial y_o}\frac{\partial y_o}{\partial x_o}\frac{\partial x_o}{\partial w_{ho}} = -(v_o - y_o)f'(x_o)y_h$$
$$\frac{\partial\varepsilon(y_0)}{\partial\theta_o} = \frac{\partial\varepsilon(y_0)}{\partial y_o}\frac{\partial y_o}{\partial x_o}\frac{\partial x_o}{\partial\theta_o} = -(v_o - y_o)f'(x_o)$$

We obtain weight deviation Δw_{ho} and bias deviation $\Delta\theta_o$ of any output unit as follows:

$$\begin{aligned}\Delta w_{ho} &= \gamma y_h(v_o - y_o)f'(x_o)\\ \Delta\theta_o &= \gamma(v_o - y_o)f'(x_o)\end{aligned} \quad (1.21)$$

Where $f'(x_o)$ is derivative of activation function $f(.)$ at x_o. Obviously,

$$\frac{\partial\varepsilon(y_0)}{\partial y_o} = -(v_o - y_o), \frac{\partial y_o}{\partial x_o} = f'(x_o), \frac{\partial x_o}{\partial w_{ho}} = y_h, \frac{\partial x_o}{\partial\theta_o} = 1$$

Let (Kröse & Smagt, 1996, p. 34),

$$\delta_0 = -\frac{\partial\varepsilon(y_0)}{\partial x_o} = -\frac{\partial\varepsilon(y_0)}{\partial y_o}\frac{\partial y_o}{\partial x_o} = (v_o - y_o)f'(x_o) \quad (1.22)$$

The quantity δ_o is called error of output unit in literature. We have the succinct equation of weight deviation Δw_{ho} and bias deviation $\Delta\theta_o$.

$$\begin{aligned}\Delta w_{ho} &= \gamma y_h\delta_o\\ \Delta\theta_o &= \gamma\delta_o\end{aligned} \quad (1.23)$$

Recall that the equation above for weigh deviation and bias deviation is derived from the squared error function $\varepsilon^{(p)}(y_o)$ at pattern p and so, it is easy to extend such equation for the total squared error function $\varepsilon(y_o) = \sum_p \varepsilon^{(p)}(y_o)$ over all patterns:

$$\Delta w_{ho} = \sum_p \Delta w_{ho}^{(p)} = \sum_p \gamma y_h^{(p)}\delta_o^{(p)}$$
$$\Delta\theta_o = \sum_p \Delta\theta_o^{(p)} = \sum_p \gamma\delta_o^{(p)}$$

The extension is easy to be asserted because the squared error function $\varepsilon^{(p)}(y_o)$ and the total squared error function $\varepsilon(y_o)$ 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:

$$w_{ho} = w_{ho} + \Delta w_{ho}$$
$$\theta_o = \theta_o + \Delta\theta_o$$

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 $\varepsilon(y_h)$ of hidden unit h is the sum of output errors $\varepsilon(y_o)$ with regard to such set of output units, as follows:

$$\varepsilon(y_h) = \sum_o \varepsilon(y_o) \tag{1.24}$$

Each output squared error $\varepsilon(y_o)$ were aforementioned:

$$\varepsilon(y_o) = \frac{1}{2}(v_o - y_o)^2$$

Note,

$$y_o = f\left(x_o = \sum_h w_{ho} y_h + \theta_o\right)$$

$$y_h = f\left(x_h = \sum_j w_{jh} y_j + \theta_h\right)$$

By default, all units are clamped and so, the clamped input s_h is the same to the input x_h as $s_h = x_h$ 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.

$$\varepsilon(y_h) = \sum_p \varepsilon^{(p)}(y_h) = \sum_p \sum_o \varepsilon^{(p)}(y_o)$$

Where,

$$\varepsilon^{(p)}(y_o) = \frac{1}{2}\left(v_o^{(p)} - y_o^{(p)}\right)^2$$

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

$$\varepsilon(y_h) = \varepsilon^{(p)}(y_h) = \sum_o \varepsilon^{(p)}(y_o) = \sum_o \varepsilon(y_o) = \sum_o \frac{1}{2}(v_o - y_o)^2$$

Recall that weight deviation Δw_{jh} and bias deviation $\Delta\theta_h$ are determined based on the gradient of the squared error function $\varepsilon(y_h)$ according to stochastic gradient descent (SGD) method for minimizing the squared error function $\varepsilon(y_h)$.

$$(w_{jh}, \theta_h) = (w_{jh}, \theta_h) - \gamma \nabla\varepsilon(w_{jh}, \theta_h)$$

Note, the gradient of $\varepsilon(y_h)$ with regard to w_{jh} and θ_h is row vector of partial derivatives of $\varepsilon(y_h)$ with regard to w_{jh} and θ_h as follows:

$$\nabla\varepsilon(y_h) = \nabla\varepsilon(w_{jh}, \theta_h) = \left(\frac{\partial\varepsilon(y_h)}{\partial w_{jh}}, \frac{\partial\varepsilon(y_h)}{\partial\theta_h}\right)$$

It is necessary to calculate the gradient $\nabla\varepsilon(w_{jh}, \theta_h)$. Firstly, we have:

$$\frac{\partial\varepsilon(y_h)}{\partial x_h} = \frac{\partial\varepsilon(y_h)}{\partial y_h}\frac{\partial y_h}{\partial x_h} = \frac{\partial\varepsilon(y_h)}{\partial y_h} f'(x_h)$$

Recall that, according to propagation rule, x_h is:

$$x_h = \sum_j w_{jh} y_j + \theta_h$$

$$y_h = f(x_h)$$

It is necessary to calculate the derivative $\frac{\partial\varepsilon(y_h)}{\partial y_h}$. Indeed, we have:

$$\frac{\partial\varepsilon(y_h)}{\partial y_h} = \sum_o \frac{\partial\varepsilon(y_o)}{\partial x_o}\frac{\partial x_o}{\partial y_h}$$

Due to:

$$\frac{\partial \varepsilon(y_o)}{\partial x_o} = -\delta_o$$

$$\frac{\partial x_o}{\partial y_h} = \frac{\partial}{\partial y_h}\left(\sum_h w_{ho}y_h + \theta_o\right) = w_{ho}$$

We obtain:

$$\frac{\partial \varepsilon(y_h)}{\partial y_h} = -\sum_o w_{ho}\delta_o$$

This implies:

$$\frac{\partial \varepsilon(y_h)}{\partial x_h} = -f'(x_h)\sum_o w_{ho}\delta_o$$

As a result, the gradient of the squared error function $\varepsilon(y_h)$ with regard to w_{jh} and θ_h is:

$$\nabla\varepsilon(y_h) = \nabla\varepsilon(w_{jh}, \theta_h) = \left(\frac{\partial \varepsilon(y_h)}{\partial w_{jh}}, \frac{\partial \varepsilon(y_h)}{\partial \theta_h}\right)$$

Where,

$$\frac{\partial \varepsilon(y_h)}{\partial w_{jh}} = \frac{\partial \varepsilon(y_h)}{\partial x_h}\frac{\partial x_h}{\partial w_{jh}} = -f'(x_h)\left(\sum_o w_{ho}\delta_o\right)y_j$$

$$\frac{\partial \varepsilon(y_h)}{\partial \theta_h} = \frac{\partial \varepsilon(y_h)}{\partial x_h}\frac{\partial x_h}{\partial \theta_h} = -f'(x_h)\sum_o w_{ho}\delta_o$$

Note,

$$\frac{\partial x_h}{\partial w_{jh}} = \frac{\partial}{\partial w_{jh}}\left(\sum_j w_{jh}y_j + \theta_h\right) = y_j$$

$$\frac{\partial x_h}{\partial \theta_h} = \frac{\partial}{\partial \theta_h}\left(\sum_j w_{jh}y_j + \theta_h\right) = 1$$

Therefore, by SGD, weight deviation Δw_{jh} and bias deviation $\Delta\theta_h$ are inversely proportional to the gradient of the squared error function $\varepsilon(y_h)$ multiplied with learning rate as follows:

$$\begin{aligned}\Delta w_{jh} &= -\gamma\frac{\partial \varepsilon(y_h)}{\partial w_{jh}} = \gamma y_j f'(x_h)\sum_o w_{ho}\delta_o \\ \Delta\theta_h &= -\gamma\frac{\partial \varepsilon(y_h)}{\partial \theta_h} = \gamma f'(x_h)\sum_o w_{ho}\delta_o\end{aligned} \quad (1.25)$$

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

$$w_{jh} = w_{jh} + \Delta w_{jh}$$
$$\theta_h = \theta_h + \Delta\theta_h$$

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:

$$\begin{aligned}\Delta w_{ho} &= \gamma y_h\delta_o \\ \Delta\theta_o &= \gamma\delta_o \\ \Delta w_{jh} &= \gamma y_j\delta_h \\ \Delta\theta_h &= \gamma\delta_h\end{aligned} \quad (1.26)$$

Where,

$$\begin{aligned}\delta_o &= (v_o - y_o)f'(x_o) \\ \delta_h &= f'(x_h)\sum_o w_{ho}\delta_o\end{aligned} \quad (1.27)$$

Note,

$$y_o = f\left(x_o = \sum_h w_{ho} y_h + \theta_o\right)$$

$$y_h = f\left(x_h = \sum_j w_{jh} y_j + \theta_h\right)$$

The quantity δ_h is called error of hidden unit in literature. The equation above is an extension of delta rule.

Recall that the equation above for weigh deviation and bias deviation is derived from the squared error function $\varepsilon^{(p)}(y_h)$ at pattern p and so, it is easy to extend such equation for the total squared error function $\varepsilon(y_h) = \sum_p \varepsilon^{(p)}(y_h)$ over all patterns:

$$\Delta w_{ho} = \sum_p \Delta w_{ho}^{(p)} = \sum_p \gamma y_h^{(p)} \delta_o^{(p)}$$

$$\Delta \theta_o = \sum_p \Delta \theta_o^{(p)} = \sum_p \gamma \delta_o^{(p)}$$

$$\Delta w_{jh} = \sum_p \Delta w_{jh}^{(p)} = \sum_p \gamma y_j^{(p)} \delta_h^{(p)}$$

$$\Delta \theta_h = \sum_p \Delta \theta_h^{(p)} = \sum_p \gamma \delta_h^{(p)}$$

Where,

$$\delta_o^{(p)} = \left(v_o^{(p)} - y_o^{(p)}\right) f'\left(x_o^{(p)}\right)$$

$$\delta_h^{(p)} = f'\left(x_h^{(p)}\right) \sum_o w_{ho}^{(p)} \delta_o^{(p)}$$

The extension is easy to be asserted because the squared error function $\varepsilon^{(p)}(y_h)$ and the total squared error function $\varepsilon(y_h)$ 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:

$$f'(x_k) = \frac{e^{-x_k}}{(1+e^{-x_k})^2} = \frac{1}{1+e^{-x_k}}\left(1 - \frac{1}{1+e^{-x_k}}\right) = f(x_k)\big(1 - f(x_k)\big) = y_k(1 - y_k)$$

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

$$\Delta w_{ho} = \gamma y_h \delta_o$$

$$\Delta \theta_o = \gamma \delta_o$$

$$\Delta w_{jh} = \gamma y_j \delta_h$$

$$\Delta \theta_h = \gamma \delta_h$$

Where,

$$\delta_o = (v_o - y_o) y_o (1 - y_o)$$

$$\delta_h = y_h(1 - y_h) \sum_o w_{ho} \delta_o \qquad (1.28)$$

Recall that δ_o and δ_h are also called errors of output unit and hidden unit, respectively.

$$Err_o = \delta_o$$

$$Err_h = \delta_h$$

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 O_i, I_j and O_j be output of unit *i*, input of unit *j*, and output of unit *j*. Obviously, we have $O_i = y_i$, $I_j = x_j = s_j$, and $O_j = y_j$. 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:

$$I_j = \sum_{i=1}^{n} w_{ij} O_i + \theta_j$$

$$O_j = \frac{1}{1 + e^{-I_j}}$$

For backpropagation algorithm, weight update rule and bias update rule of any unit *j* are represented as follows:

$$\Delta w_{ij} = \gamma O_i Err_j$$

$$\Delta \theta_o = \gamma Err_j$$

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

$$Err_j = \begin{cases} (V_j - O_j)O_j(1 - O_j) \text{ for output unit } j \\ O_j(1 - O_j)\sum_k w_{jk} Err_k \text{ for hidden unit } j \end{cases}$$

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 $w_{ij}$ of all connections between units are initialized as random real numbers which should be in space [0, 1]. Each bias $\theta_i$ 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 $I_i$ and its output value denoted $O_i$ are the same.
            $O_i = I_i$
        End for each input unit $i$
        For each hidden unit $j$ or output unit $j$, its input value $I_j$ is the weighted sum of all output values of units from previous layer. The bias is also added to this weighted sum.
            $I_j = \sum_i w_{ij} O_i + \theta_j$
            Where $w_{ij}$ is the weight of connection from unit $i$ in previous layer to unit $j$, $O_i$ is output value of unit $i$ from previous layer and $\theta_j$ is bias of unit $j$. The output value of hidden unit or output unit $O_j$ is computed by applying activation function to its input value (weighted sum). Suppose activation function is sigmoid function. We have:
            $O_j = \frac{1}{1 + e^{-I_j}}$
        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 $Err_j$ is computed as below:
            $Err_j = O_j(1 - O_j)(V_j - O_j)$
            Where $V_j$ is the real value of unit $j$ in training corpus; in other words, $V_j$ is the actual class. This error is the $\delta_o$ 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:
            $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$
            Where $w_{jk}$ is the weight of the connection from hidden unit $j$ to a unit $k$ in next higher layer and $Err_k$ is the error of unit $k$. This error is the $\delta_h$ aforementioned.
        End for each hidden unit $j$

        4. Updating weights and biases is based on the errors.
        For each weight $w_{ij}$ over the whole NN. The weights are updated so as to minimize the errors. Given $\Delta w_{ij}$ is the change in weight $w_{ij}$, the weight $w_{ij}$ is updated as below:
            $\Delta w_{ij} = \gamma * Err_j O_i$
            $w_{ij} = w_{ij} + \Delta w_{ij}$
            Where $\gamma$ 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 $w_{ij}$ in the whole NN
        For each bias $\theta_j$ over the whole NN. The bias $\theta_j$ of hidden or output unit $j$ is updated
        as below:
                    $\Delta\theta_j = \gamma * Err_j$
                    $\theta_j = \theta_j + \Delta\theta_j$
            Where $\gamma$ is learning rate ranging from 0 to 1 ($0 < \gamma \leq 1$).
        End for each bias $\theta_j$

    End for each data row in corpus
End while terminating condition is not satisfied with note that there are two common
terminating conditions:
    - All $\Delta w_{ij}$ 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.

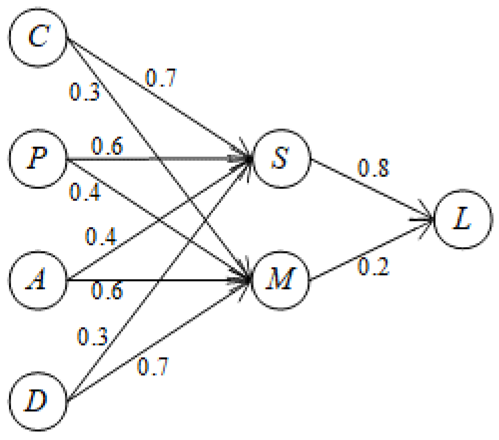

Figure 1.4. The NN for document classification

Note that units C, P, A and D denote terms "*computer*", "*programming language*", "*algorithm*", and "*derivative*", respectively. Units S and M denote "*computer science*" class and "*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 $\mathcal{D}$ = {*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:

$$tf_{11} = \frac{tf_{11}}{tf_{11} + tf_{12} + tf_{13} + tf_{14}} = \frac{5}{5+3+1+1} = 0.5$$

$$tf_{12} = \frac{tf_{12}}{tf_{11} + tf_{12} + tf_{13} + tf_{14}} = \frac{3}{5+3+1+1} \approx 0.3$$

$$tf_{13} = \frac{tf_{13}}{tf_{11} + tf_{12} + tf_{13} + tf_{14}} = \frac{1}{5+3+1+1} = 0.1$$

$$tf_{14} = \frac{tf_{14}}{tf_{11} + tf_{12} + tf_{13} + tf_{14}} = \frac{1}{5+3+1+1} = 0.1$$

Following table shows normalized term frequencies in corpus $\mathcal{D}$.

	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 I_C, I_P, I_A, I_D, I_S, I_M, and I_L be input values of units *C*, *P*, *A*, *D*, *S*, *M*, and *L*. Let O_C, O_P, O_A, O_D, O_S, O_M, and O_L be output values of units *C*, *P*, *A*, *D*, *S*, *M*, and *L*. Let θ_S, θ_M, and θ_L be biases of units *S*, *M*, and *L*. Suppose all biases are initialized by zero, we have θ_S=θ_M=θ_L=0. Let w_{CS}, w_{CM}, w_{PS}, w_{PM}, w_{AS}, w_{AM}, w_{DS}, w_{DM}, w_{SL}, and w_{ML} 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 w_{CS}=0.7, w_{CM}=0.3, w_{PS}=0.6, w_{PM}=0.4, w_{AS}=0.4, w_{AM}=0.6, w_{DS}=0.3, w_{DM}=0.7, w_{SL}=0.8, and w_{ML}=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 O_S, O_M, O_L and update connection weights. For simplicity, activation function is sigmoid function $f(x) = \frac{1}{1+e^{-x}}$. According to propagation rule (Han & Kamber, 2006, p. 331) for computing output value of a unit, we have:

O_C=I_C=0.5
O_P=I_P=0.3
O_A=I_A=0.1
O_D=I_D=0.1

$$I_S = w_{CS}O_C + w_{PS}O_P + w_{AS}O_A + w_{DS}O_D + \theta_s$$
$$= 0.7 * 0.5 + 0.6 * 0.3 + 0.4 * 0.1 + 0.3 * 0.1 + 0 = 0.6$$
$$O_S = \mu(I_S) = \frac{1}{1 + \exp(-I_S)} = \frac{1}{1 + \exp(-0.6)} \approx 0.65$$
$$I_M = w_{CM}O_C + w_{PM}O_P + w_{AM}O_A + w_{DM}O_D + \theta_M$$
$$= 0.3 * 0.5 + 0.4 * 0.3 + 0.6 * 0.1 + 0.7 * 0.1 + 0 = 0.4$$
$$O_M = \mu(I_M) = \frac{1}{1 + \exp(-I_M)} = \frac{1}{1 + \exp(-0.4)} \approx 0.6$$
$$I_L = w_{SL}O_S + w_{ML}O_M + \theta_L = 0.8 * 0.65 + 0.2 * 0.6 + 0 \approx 0.64$$
$$O_L = \frac{1}{1 + \exp(-I_L)} = \frac{1}{1 + \exp(-0.64)} \approx 0.65$$

Let V_L be value of output unit L. Because D_1 belongs to "*computer science*" class, we have:

$$V_L = 1$$

Let Err_L, Err_S, and Err_M be errors of units L, S, and M, respectively. According to the equation for updating error of output unit, we have:

$$Err_L = O_L(1 - O_L)(V_L - O_L) = 0.65 * (1 - 0.65) * (1 - 0.65) \approx 0.08$$

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

$$Err_S = O_S(1 - O_S)Err_LW_{SL} = 0.65 * (1 - 0.65) * 0.08 * 0.8 \approx 0.01$$
$$Err_M = O_M(1 - O_M)Err_LW_{ML} = 0.6 * (1 - 0.6) * 0.08 * 0.2 \approx 0$$

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

$$w_{CS} = w_{CS} + \Delta w_{CS} = w_{CS} + 1 * Err_SO_C = 0.7 + 1 * 0.01 * 0.5 \approx 0.71$$
$$w_{CM} = w_{CM} + \Delta w_{CM} = w_{CM} + 1 * Err_MO_C = 0.3 + 1 * 0 * 0.5 \approx 0.3$$
$$w_{PS} = w_{PS} + \Delta w_{PS} = w_{PS} + 1 * Err_SO_P = 0.6 + 1 * 0.01 * 0.3 \approx 0.6$$
$$w_{PM} = w_{PM} + \Delta w_{PM} = w_{PM} + 1 * Err_MO_P = 0.4 + 1 * 0 * 0.3 \approx 0.4$$
$$w_{AS} = w_{AS} + \Delta w_{AS} = w_{AS} + 1 * Err_SO_A = 0.4 + 1 * 0.01 * 0.1 \approx 0.4$$
$$w_{AM} = w_{AM} + \Delta w_{AM} = w_{AM} + 1 * Err_MO_A = 0.6 + 1 * 0 * 0.1 \approx 0.6$$
$$w_{DS} = w_{DS} + \Delta w_{DS} = w_{DS} + 1 * Err_SO_D = 0.3 + 1 * 0.01 * 0.1 \approx 0.3$$
$$w_{DM} = w_{DM} + \Delta w_{DM} = w_{DM} + 1 * Err_MO_D = 0.7 + 1 * 0 * 0.1 \approx 0.7$$
$$w_{SL} = w_{SL} + \Delta w_{SL} = w_{SL} + 1 * Err_LO_S = 0.8 + 1 * 0.08 * 0.65 \approx 0.85$$
$$w_{ML} = w_{ML} + \Delta w_{ML} = w_{ML} + 1 * Err_LO_M = 0.2 + 1 * 0.08 * 0.6 \approx 0.25$$

According to the equation for updating biases θ_S, θ_M, and θ_L, we have:

$$\theta_S = \theta_S + \Delta\theta_S = \theta_S + 1 * Err_S = 0 + 1 * 0.01 = 0.01$$
$$\theta_M = \theta_M + \Delta\theta_M = \theta_M + 1 * Err_M = 0 + 1 * 0 = 0$$
$$\theta_L = \theta_L + \Delta\theta_L = \theta_L + 1 * Err_L = 0 + 1 * 0.08 = 0.08$$

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 O_S, O_M, O_L and update final connection weights. The following table shows results from this training process based on backpropagation algorithm.

	Inputs	Outputs	Weights	Biases
D_1	I_C=0.5 I_P=0.3	O_S=0.65 O_M=0.60	w_{CS}=0.70 w_{CM}=0.30	θ_S=0.01 θ_M=0.00

	I_A=0.1 I_D=0.1	O_L=0.65	w_{PS}=0.60 w_{PM}=0.40 w_{AS}=0.40 w_{AM}=0.60 w_{DS}=0.30 w_{DM}=0.70 w_{SL}=0.85 w_{ML}=0.25	θ_L=0.08
D_2	I_C=0.05 I_P=0.05 I_A=0.40 I_D=0.50	O_S=0.60 O_M=0.65 O_L=0.71	w_{CS}=0.70 w_{CM}=0.30 w_{PS}=0.60 w_{PM}=0.40 w_{AS}=0.39 w_{AM}=0.59 w_{DS}=0.29 w_{DM}=0.69 w_{SL}=0.76 w_{ML}=0.40	θ_S=–0.02 θ_M=–0.01 θ_L=–0.07
D_3	I_C=0.05 I_P=0.05 I_A=0.40 I_D=0.50	O_S=0.60 O_M=0.64 O_L=0.67	w_{CS}=0.70 w_{CM}=0.30 w_{PS}=0.60 w_{PM}=0.40 w_{AS}=0.38 w_{AM}=0.59 w_{DS}=0.27 w_{DM}=0.68 w_{SL}=0.68 w_{ML}=0.41	θ_S=–0.04 θ_M=–0.03 θ_L=–0.22
D_4	I_C=0.20 I_P=0.05 I_A=0.20 I_D=0.55	O_S=0.62 O_M=0.60 O_L=0.62	w_{CS}=0.70 w_{CM}=0.30 w_{PS}=0.61 w_{PM}=0.41 w_{AS}=0.38 w_{AM}=0.59 w_{DS}=0.27 w_{DM}=0.68 w_{SL}=0.73 w_{ML}=0.55	θ_S=–0.03 θ_M=–0.02 θ_L=–0.13
D_5	I_C=0.15 I_P=0.15 I_A=0.40 I_D=0.30	O_S=0.60 O_M=0.63 O_L=0.65	w_{CS}=0.70 w_{CM}=0.30 w_{PS}=0.61 w_{PM}=0.40 w_{AS}=0.37 w_{AM}=0.58 w_{DS}=0.27 w_{DM}=0.68 w_{SL}=0.64 w_{ML}=0.41	θ_S=–0.05 θ_M=–0.04 θ_L=–0.28
D_6	I_C=0.35 I_P=0.10	O_S=0.61 O_M=0.61	w_{CS}=0.70 w_{CM}=0.30	θ_S=–0.04 θ_M=–0.03

	I_A=0.45 I_D=0.10	O_L=0.60	w_{PS}=0.61 w_{PM}=0.40 w_{AS}=0.38 w_{AM}=0.59 w_{DS}=0.27 w_{DM}=0.68 w_{SL}=0.70 w_{ML}=0.56	θ_L=–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.

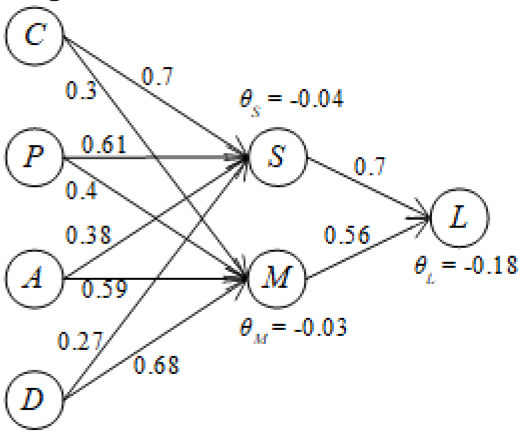

Figure 1.5. Trained neural network

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 w_{CS}=0.7, w_{CM}=0.3, w_{PS}=0.61, w_{PM}=0.4, w_{AS}=0.38, w_{AM}=0.59, w_{DS}=0.27, w_{DM}=0.68, w_{SL}=0.7, w_{ML}=0.56 and biases θ_S=–0.04, θ_M=–0.03, θ_L=–0.18. It is required to compute the output values O_S, O_M, and O_L. For simplicity, activation function is sigmoid function $\mu(x)=\frac{1}{1+e^{-x}}$. According to the equation (Han & Kamber, 2006, p. 331) for computing the output value of a unit, we have:

$$I_S = w_{CS}O_C + w_{PS}O_P + w_{AS}O_A + w_{DS}O_D + \theta_S$$
$$= 0.7 * 0.4 + 0.61 * 0.3 + 0.38 * 0.1 + 0.27 * 0.2 - 0.04 \approx 0.52$$
$$O_S = \mu(I_S) = \frac{1}{1+\exp(-I_S)} = \frac{1}{1+\exp(-0.52)} \approx 0.63$$
$$I_M = w_{CM}O_C + w_{PM}O_P + w_{AM}O_A + w_{DM}O_D + \theta_M$$
$$= 0.3 * 0.4 + 0.4 * 0.3 + 0.59 * 0.1 + 0.68 * 0.2 - 0.03 \approx 0.41$$
$$O_M = \mu(I_M) = \frac{1}{1+\exp(-I_M)} = \frac{1}{1+\exp(-0.41)} \approx 0.6$$
$$I_L = w_{SL}O_S + w_{SM}O_M + \theta_L = 0.7 * 0.63 + 0.56 * 0.6 - 0.18 \approx 0.6$$
$$O_L = \frac{1}{1+\exp(-I_L)} = \frac{1}{1+\exp(-0.6)} \approx 0.65$$

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

2. Convergence of learning algorithm

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

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

$$w_{jk} = x_j x_k$$

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

$$\Delta w_{jk} = \gamma y_j y_k$$

Note, the positive constant γ which is called learning rate specifies the power of proportional whereas y_j and y_k are outputs of unit j and unit k. Of course, weight deviation Δw_{jk} 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) x_i like aforementioned Perceptron. Suppose bias is zero, propagation rule is:

$$y = \sum_{i=1}^{n} w_i x_i$$

We will study the convergence of the following Hebbian rule for learning weight vector $\boldsymbol{w}$ = $(w_1, w_2,\ldots, w_n)^T$ with $\boldsymbol{x} = (x_1, x_2,\ldots, x_n)^T$.

$$w_i = w_i + \Delta w_i = w_i + x_i y$$

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

$$w_i(t) = t x_i y$$

Where,

$$y = (\boldsymbol{w}^*)^T\boldsymbol{x} = \sum_{i=1}^{n} w_i^* x_i$$

So, we have:

$$\boldsymbol{w}(t) = ty\boldsymbol{x}$$

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

$$\cos(\boldsymbol{w}(t), \boldsymbol{w}^*) = \frac{ty\boldsymbol{x}^T\boldsymbol{w}^*}{\sqrt{ty}|\boldsymbol{x}||\boldsymbol{w}^*|} = \sqrt{ty}\frac{\boldsymbol{x}^T\boldsymbol{w}^*}{|\boldsymbol{x}||\boldsymbol{w}^*|} = \sqrt{t}\frac{(\boldsymbol{x}^T\boldsymbol{w}^*)^{\frac{3}{2}}}{|\boldsymbol{x}||\boldsymbol{w}^*|}$$

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

$$\lim_{t\to\infty} \cos(\boldsymbol{w}(t), \boldsymbol{w}^*) = +\infty > 1$$

Therefore, the iterative learning algorithm must stop at some finite *t* iterations with the optimal weight $\boldsymbol{w}^*$. 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 $\boldsymbol{w}^*$. So, we need to research the delta rule which is an improved version of Hebbian rule.

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

$$\Delta w_{jk} = \gamma y_j \delta_k$$
$$\Delta \theta_k = \gamma \delta_k$$

Where,

$$\delta_k = \begin{cases} (v_k - y_k)f'(x_k) \text{ for ouput unit} \\ f'(x_k)\sum_l w_{kl}\delta_l \text{ for hidden unit} \end{cases}$$

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 $f(\boldsymbol{w})$, GD is an iterative algorithm that moves the parameter $\boldsymbol{w}$ along descending direction which is the opposite of gradient of $f(\boldsymbol{w})$ at every time point (or iteration) *t* until reaching the optimizer $\boldsymbol{w}^*$.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma_t \nabla f(\boldsymbol{w}_t)$$

Note, γ_t is length of descending direction at time point *t*, which is also called learning rate. Moreover, $f(\boldsymbol{w})$ receives some data $\boldsymbol{x}$ as input.

$$f(\boldsymbol{w}) = f(\boldsymbol{w}|\boldsymbol{x})$$

For learning NN with weight update rule and bias update rule, $f(\boldsymbol{w})$ is the squared error function $\varepsilon(.)$ whose parameters are weights. In general case $\boldsymbol{w}$ is vector. When $f(\boldsymbol{w})$ is averaged sum of a large number of member target functions $f_i(\boldsymbol{w}_i)$ (De Sa, 2021, p. 1):

$$f(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{w}_i)$$

Where $\boldsymbol{w}$ is composed of many parts as $\boldsymbol{w} = (\boldsymbol{w}_1, \boldsymbol{w}_2,\ldots, \boldsymbol{w}_n)^T$. However, without loss of generality, we can denote $f_i(\boldsymbol{w})$ by convention that $f_i(\boldsymbol{w})$ only acts on its part $\boldsymbol{w}_i$ while considering other parts $\boldsymbol{w}_j$ where $j \neq i$ as constants or ignoring them in its analytic formulation, as follows:

$$f(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{w}) \tag{2.1}$$

Anyhow, an important aspect is that the gradient of $f(\boldsymbol{w})$ is always averaged sum of gradients of all $f_i(\boldsymbol{w})$ as follows:

$$\nabla f(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{w}) \quad (2.2)$$

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 $\nabla f_i(\boldsymbol{w})$. Suppose there is a sample $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N, \ldots\}$ where $\boldsymbol{x}_i$ is corresponding to some $f_k(.)$, SGD will feed these $\boldsymbol{x}_i$ (s) one by one or batch by batch (De Sa, 2021, p. 1) for each time point *t* to learn ***w***.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma_t \nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t) \quad (2.3)$$

Where $f_{\hat{\imath}_t}(.)$ is some $f_k(.)$ corresponding to the data $\boldsymbol{x}_i$ in the sample. For instance, if $\hat{\imath}_t = k$ given data point $\boldsymbol{x}_i$ at time point *t* then, $\boldsymbol{x}_i$ will be fed to the member function $f_k(\boldsymbol{w}_t) = f_k(\boldsymbol{w}_t \mid \boldsymbol{x}_i)$ at time point *t*. Moreover, if $\boldsymbol{x}_i$ is fed to a set of *m* member functions, for example $\{f_1(), f_2(.), \ldots., f_m(.)\}$ at one time then, it is possible to consider that $\boldsymbol{x}_i$ is fed *m* times, each time point for one member function, without loss of generality. Because $\hat{\imath}_t$ is selected among *n* member functions $f_i(\boldsymbol{w})$, probability distribution of $\hat{\imath}_t$ is even as follows (De Sa, 2021, p. 2):

$$P(\hat{\imath}_t) = \frac{1}{n}, \forall \hat{\imath}_t$$

This probability distribution is called *selective distribution*. It is more important that $\boldsymbol{w}_t$ follows a so-called *stochastic distribution* below:

$$\boldsymbol{w}_t \sim g(\boldsymbol{w}_t)$$

The stochastic distribution $g(\boldsymbol{w}_t)$ implies $\boldsymbol{w}_t$ is moved randomly because data $\boldsymbol{x}_i$ 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 $f_i(\boldsymbol{w})$ at one providing time. Besides, the iterative feeding process makes SGD feasible to calculate a gradient $\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)$ with some data $\boldsymbol{x}_i$ (or package $\boldsymbol{x}_i$) at one time.

In order to assure the convergence of SGD, we need to research Lipschitz continuity. Recall that if function $f_i(.)$ is Lipschitz continuous then, given any two vector $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ we have (Wikipedia, Lipschitz continuity, 2001):

$$\|f_i(\boldsymbol{w}_1) - f_i(\boldsymbol{w}_2)\| \le L_i \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|$$

Where L_i 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:

$$\|f_i(\boldsymbol{w})\| \le L_i \|\boldsymbol{w}\| \text{ or } \|f_i(\boldsymbol{w})\|^2 \le L\|\boldsymbol{w}\|^2$$

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

$$\begin{cases} \|f_i(\boldsymbol{w}_1) - f_i(\boldsymbol{w}_2)\| \le L_i \|\boldsymbol{w}_1 - \boldsymbol{w}_2\| \\ \|\nabla f_i(\boldsymbol{w})\| \le G_i \end{cases}, \forall i, \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w} \quad (2.4)$$

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

$$\begin{cases} \|f_i(\boldsymbol{w}_1) - f_i(\boldsymbol{w}_2)\| \le L_i \|\boldsymbol{w}_1 - \boldsymbol{w}_2\| \\ \|\nabla f_i(\boldsymbol{w})\| \le G \end{cases}, \forall i, \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}$$

The bounded condition of gradient $\|\nabla f_i(\boldsymbol{w})\| \le G$ is not strict because we can restrict magnitude of this gradient when implementing SGD, for example, $\nabla f_i(\boldsymbol{w})$ is normalized as follows:

$$\nabla f_i(\boldsymbol{w}) = \frac{\nabla f_i(\boldsymbol{w})}{|\nabla f_i(\boldsymbol{w})|}$$

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 $\nabla f_i(\boldsymbol{w})$ is Lipschitz continuous, its derivative $\nabla^2 f_i(\boldsymbol{w})$ which is Hessian matrix (second-order derivative) of $f_i(\boldsymbol{w})$ is bounded according to the important property, as follows:

$$\|\nabla^2 f_i(\boldsymbol{w})\| \le H_i, \forall i, \boldsymbol{w} \tag{2.5}$$

Where H_i is a constant. When $\nabla^2 f_i(\boldsymbol{w})$ 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 $f_i(\boldsymbol{w})$ and their gradients $\nabla f_i(\boldsymbol{w})$ are Lipschitz continuous*. The bounding of $\nabla^2 f_i(\boldsymbol{w})$ as $\|\nabla^2 f_i(\boldsymbol{w})\| \le H_i$ derives (De Sa, 2021, p. 2):

$$\|\boldsymbol{w}^T \nabla^2 f_i(\boldsymbol{w}) \boldsymbol{w}\| \le \|\boldsymbol{w}^T\| \|\nabla^2 f_i(\boldsymbol{w})\| \|\boldsymbol{w}\| \le H_i \|\boldsymbol{w}\|^2$$

Suppose Hessian matrix $\nabla^2 f_i(\boldsymbol{w})$ is a set of basic vectors of a vector space that is image of Euclidean space, hence, $\nabla^2 f_i(\boldsymbol{w})$ represents a mapping with note that $|\boldsymbol{w}^T \nabla^2 f_i(\boldsymbol{w}) \boldsymbol{w}|$ is square of the norm of $\boldsymbol{w}$ in the vector space specified by $\nabla^2 f_i(\boldsymbol{w})$ whereas $|\boldsymbol{w}|^2$ is square of the norm of $\boldsymbol{w}$ in Euclidean space. In other words, here $\nabla^2 f_i(\boldsymbol{w})$ shrinks vector space. Obviously, we also have:

$$\|\boldsymbol{w}^T \nabla^2 f(\boldsymbol{w}) \boldsymbol{w}\| \le H \|\boldsymbol{w}\|^2$$

Where H is a constant too, due to:

$$\|\boldsymbol{w}^T \nabla^2 f(\boldsymbol{w}) \boldsymbol{w}\| = \left\| \boldsymbol{w}^T \frac{1}{n} \sum_{i=1}^{n} \nabla^2 f_i(\boldsymbol{w})\, \boldsymbol{w} \right\| = \frac{1}{n} \left\| \sum_{i=1}^{n} \boldsymbol{w}^T \nabla^2 f_i(\boldsymbol{w}) \boldsymbol{w} \right\| \le \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{w}^T \nabla^2 f_i(\boldsymbol{w}) \boldsymbol{w}\|$$
$$\le \left(\frac{1}{n} \sum_{i=1}^{n} L_i \right) \|\boldsymbol{w}\|^2 = H \|\boldsymbol{w}\|^2$$

Where let,

$$H = \frac{1}{n} \sum_{i=1}^{n} H_i$$

Recall that SGD is an iterative algorithm which feeds data $\boldsymbol{x}_i$ (s) one by one or batch by batch (De Sa, 2021, p. 1) for each time point t to learn $\boldsymbol{w}$.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma_t \nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)$$

In order to prove the convergence of SGD, we need to prove that the expectation of norm of the stochastic gradient $\nabla f(\boldsymbol{w}_t)$ 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 $\nabla f(\boldsymbol{w}_t)$ is zero with note that the expectation is associated with the stochastic distribution $g(\boldsymbol{w}_t)$ and selective distribution $P(\hat{\imath}_t)$. In general, we will prove the equation as follows:

$$\lim_{t \to \infty} E(\|\nabla f(\boldsymbol{w}_t)\|) = 0 \tag{2.6}$$

Or,

$$\lim_{t \to \infty} E(\|\nabla f(\boldsymbol{w}_t)\|^2) = 0$$

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 $f(\boldsymbol{w}_{t+1})$ at $\boldsymbol{w}_t$ according to Taylor's theorem, there is a ξ_t between $\boldsymbol{w}_t$ and $\boldsymbol{w}_{t+1}$ such that (De Sa, 2021, p. 2):

$$f(\boldsymbol{w}_{t+1}) = f\left(\boldsymbol{w}_t - \gamma_t \nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)$$
$$= f(\boldsymbol{w}_t) - \left(\gamma_t \nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) + \frac{1}{2}\left(\gamma_t \nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla^2 f(\xi_t) \left(\gamma_t \nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)$$

$$\leq f(\boldsymbol{w}_t) - \gamma_t \left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) + \frac{\gamma_t^2 H}{2} \left\|\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right\|^2$$

(Due to $\|\boldsymbol{w}^T \nabla^2 f(\boldsymbol{w}) \boldsymbol{w}\| \leq H\|\boldsymbol{w}\|^2$)

$$\leq f(\boldsymbol{w}_t) - \gamma_t \left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) + \frac{\gamma_t^2 G^2 H}{2}$$

(Due to $\|\nabla f_i(\boldsymbol{w})\| \leq G$)

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

$$\gamma_t \left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) \leq f(\boldsymbol{w}_t) - f(\boldsymbol{w}_{t+1}) + \frac{\gamma_t^2 G^2 H}{2}$$

Taking expectation on both sides of the inequation above by both stochastic distribution $g(\boldsymbol{w}_t)$ and selective distribution $P(\hat{\imath}_t)$, we have:

$$\gamma_t E\left(\left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) \middle| g(\boldsymbol{w}_t), P(\hat{\imath}_t)\right) \leq E\left(f(\boldsymbol{w}_t) - f(\boldsymbol{w}_{t+1}) \middle| g(\boldsymbol{w}_t), P(\hat{\imath}_t)\right) + \frac{\gamma_t^2 G^2 H}{2}$$

Please pay attention that γ_t is independent from both stochastic distribution $g(\boldsymbol{w}_t)$ and selective distribution $P(\hat{\imath}_t)$. Because $f(\boldsymbol{w}_t)$ and $f(\boldsymbol{w}_{t+1})$ are independent from the selective distribution $P(\hat{\imath}_t)$, we have:

$$\gamma_t E\left(\left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) \middle| g(\boldsymbol{w}_t), P(\hat{\imath}_t)\right) \leq E\left(f(\boldsymbol{w}_t) - f(\boldsymbol{w}_{t+1}) \middle| g(\boldsymbol{w}_t)\right) + \frac{\gamma_t^2 G^2 H}{2}$$

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

$$P(\hat{\imath}_t) = \frac{1}{n}, \forall \hat{\imath}_t$$

We have:

$$E\left(\left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) \middle| g(\boldsymbol{w}_t), P(\hat{\imath}_t)\right) = \int_{\boldsymbol{x}_t} \sum_{i=1}^{n} P(\hat{\imath}_t = i) \left(\left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) g(\boldsymbol{w}_t) d\boldsymbol{w}_t\right)$$

$$= \int_{\boldsymbol{x}_t} \left(\sum_{i=1}^{n} P(\hat{\imath}_t = i) \left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T\right) \nabla f(\boldsymbol{w}_t) g(\boldsymbol{w}_t) d\boldsymbol{w}_t$$

$$= \int_{\boldsymbol{x}_t} \left(\frac{1}{n}\sum_{i=1}^{n} \left(\nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)^T\right) \nabla f(\boldsymbol{w}_t) g(\boldsymbol{w}_t) d\boldsymbol{w}_t = \int_{\boldsymbol{x}_t} \left(\nabla f(\boldsymbol{w}_t)\right)^T \nabla f(\boldsymbol{w}_t) g(\boldsymbol{w}_t) d\boldsymbol{w}_t$$

$$\left(\text{Due to } \nabla f(\boldsymbol{w}_t) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_{\hat{\imath}_t}(\boldsymbol{w}_t)\right)$$

$$= \int_{\boldsymbol{x}_t} \|\nabla f(\boldsymbol{w}_t)\|^2 g(\boldsymbol{w}_t) d\boldsymbol{w}_t = E\left(\|\nabla f(\boldsymbol{w}_t)\|^2 \middle| g(\boldsymbol{w}_t)\right)$$

This implies:

$$\gamma_t E\left(\|\nabla f(\boldsymbol{w}_t)\|^2 \middle| g(\boldsymbol{w}_t)\right) \leq E\left(f(\boldsymbol{w}_t) - f(\boldsymbol{w}_{t+1}) \middle| g(\boldsymbol{w}_t)\right) + \frac{\gamma_t^2 G^2 H}{2}$$

As a convention, $g(\boldsymbol{w}_t)$ is the default distribution and so it is implied in the expectation and so we can denote:

$$\gamma_t E(\|\nabla f(\boldsymbol{w}_t)\|^2) \leq E\left(f(\boldsymbol{w}_t) - f(\boldsymbol{w}_{t+1})\right) + \frac{\gamma_t^2 G^2 H}{2}$$

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

$$\sum_{t=0}^{T-1}\gamma_t E(\|\nabla f(\boldsymbol{w}_t)\|^2) \le \sum_{t=0}^{T-1} E\big(f(\boldsymbol{w}_t) - f(\boldsymbol{w}_{t+1})\big) + \frac{G^2H}{2}\sum_{t=0}^{T-1}\gamma_t^2$$
$$= f(\boldsymbol{w}_0) - f(\boldsymbol{w}_T) + \frac{G^2H}{2}\sum_{t=0}^{T-1}\gamma_t^2$$

Suppose the optimization problem is minimization problem, let f^* is the expected optimal value such that $f^* \le f(\boldsymbol{w}_T)$ for all T, we have (De Sa, 2021, p. 2):

$$\sum_{t=0}^{T-1}\gamma_t E(\|\nabla f(\boldsymbol{w}_t)\|^2) \le f(\boldsymbol{w}_0) - f^* + \frac{G^2H}{2}\sum_{t=0}^{T-1}\gamma_t^2$$

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

$$P(\tau = t) = \frac{\gamma_t}{\sum_{k=0}^{T-1}\gamma_k}$$

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

$$E(\|\nabla f(\boldsymbol{w}_\tau)\|^2) = \sum_{t=0}^{T-1} E(\|\nabla f(\boldsymbol{w}_t)\|^2)P(\tau = t) = \frac{1}{\sum_{k=0}^{T-1}\gamma_k}\sum_{t=0}^{T-1}\gamma_t E(\|\nabla f(\boldsymbol{w}_t)\|^2)$$

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

$$E(\|\nabla f(\boldsymbol{w}_\tau)\|^2) \le \frac{1}{\sum_{t=0}^{T-1}\gamma_t}\left(f(\boldsymbol{w}_0) - f^* + \frac{G^2H}{2}\sum_{t=0}^{T-1}\gamma_t^2\right) \tag{2.7}$$

If fixing learning rate such that $\gamma_t = \gamma$, we have (De Sa, 2021, p. 3):

$$E(\|\nabla f(\boldsymbol{w}_\tau)\|^2) \le \frac{f(\boldsymbol{w}_0) - f^*}{T\gamma} + \frac{\gamma G^2H}{2}$$

Due to:

$$\lim_{\tau\to\infty}\left(\frac{f(\boldsymbol{w}_0) - f^*}{T\gamma} + \frac{\gamma G^2H}{2}\right) = \lim_{T\to\infty}\left(\frac{f(\boldsymbol{w}_0) - f^*}{T\gamma} + \frac{\gamma G^2H}{2}\right) = \frac{\gamma G^2H}{2} \ne 0$$

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

$$\gamma_t = \frac{1}{\sqrt{t+1}} \tag{2.8}$$

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

$$\sum_{t=0}^{T-1}\gamma_t = \sum_{t=0}^{T-1}\frac{1}{\sqrt{t+1}} \cong \int_0^T \frac{1}{\sqrt{x}}dx = 2\sqrt{T}$$
$$\sum_{t=0}^{T-1}\gamma_t^2 = \sum_{t=0}^{T-1}\frac{1}{t+1} \cong \int_0^T \frac{1}{x}dx = \log(T+1)$$

We have:

$$0 \le E(\|\nabla f(\boldsymbol{w}_\tau)\|^2) \le \frac{2(f(\boldsymbol{w}_0) - f^*) + G^2H\log(T+1)}{4\sqrt{T}} = \mathcal{O}\left(\frac{1}{\sqrt{T}}\right) \tag{2.9}$$

Due to:

$$\lim_{\tau\to\infty}\left(\frac{2(f(\boldsymbol{w}_0) - f^*) + G^2H\log(T+1)}{4\sqrt{T}}\right) = \lim_{T\to\infty}\mathcal{O}\left(\frac{1}{\sqrt{T}}\right) = 0$$

We obtain:

$$\lim_{\tau \to \infty} E(\|\nabla f(\boldsymbol{w}_\tau)\|^2) = 0$$

As a result, we assert that SGD will converge if all member functions $f_i(\boldsymbol{w})$ and their gradients $\nabla^2 f_i(\boldsymbol{w})$ 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.

3. 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):

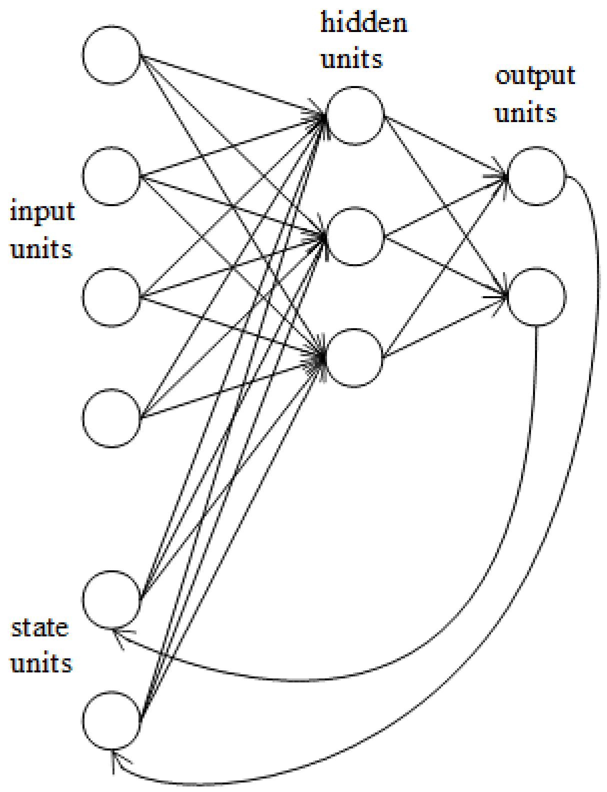

Figure 3.1. Jordan network

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):

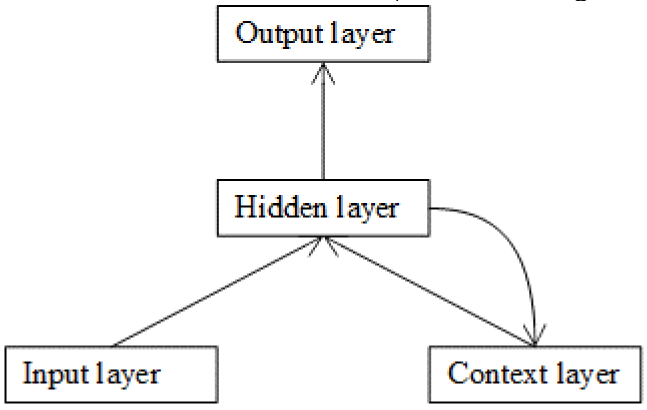

Figure 3.2. Elman network

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.

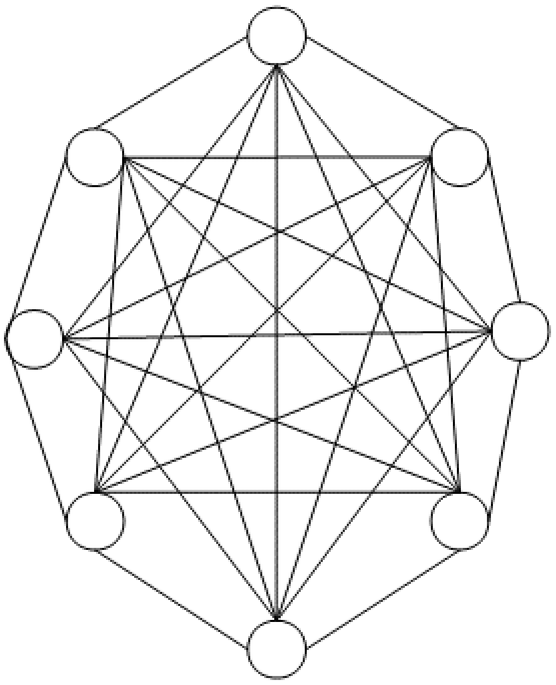

Figure 3.3. Hopfield network

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):

$$x_k(t+1) = \sum_{j \neq k} w_{jk} y_j(t) + \theta_k$$

$$y_k(t+1) = f\big(x_k(t+1)\big) = \begin{cases} +1 \text{ if } x_k(t+1) > U_k \\ -1 \text{ if } x_k(t+1) < U_k \\ y_k(t) \text{ otherwise} \end{cases} \quad (3.1)$$

Where U_k 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:

$$x_k = \sum_{j \neq k} w_{jk} y_j + \theta_k$$

$$y_k = f(x_k) = \begin{cases} +1 \text{ if } x_k > U_k \\ -1 \text{ if } x_k < U_k \\ y_k (\text{not changed}) \text{ otherwise} \end{cases}$$

Suppose there are *n* units, weights in Hopfield network form a square $n_x n$ weight matrix $W = (w_{ij})_{nxn}$ with convention that $w_{ii} = 0$ which implies that a unit does not connect with itself.

$$W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{pmatrix}$$

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

$$\Theta = (\theta_1, \theta_2, \dots, \theta_n)^T$$

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

$$y_k(t) = y_k(t-1) \quad (3.2)$$

If time point is not concerned, a unit *k* is stable if its y_k 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 $\boldsymbol{x} = (x_1, x_2, \dots, x_n)^T$, 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 $\boldsymbol{x} = (x_1, x_2, \dots, x_n)^T$ of $n$ units, weight matrix $W$ is initialized arbitrarily
with suppose $W$ is symmetric, and bias vector $\Theta$ is initialized as zero vector $\Theta = \boldsymbol{0}^T$.
Output: weight matrix $W$ and biases vector $\Theta$ are trained at which all units are stable.

All outputs are initialized by inputs such that $y_k = x_k$ for all $k$.
Repeat
    Calculate biases $\theta_k$ and outputs $y_k$ of all units according to Bruce algorithm (Kröse &
    Smagt, 1996, p. 52) and propagation rule as follows:
        $\theta_k = \begin{cases} 0 \text{ if } y_k \text{ is stable} \\ 1 \text{ otherwise} \end{cases}$
        $s_k = \sum_{j \neq k} w_{jk} y_j + \theta_k$
        $y_k = \begin{cases} +1 \text{ if } s_k > U_k \\ -1 \text{ if } s_k < U_k \\ y_k (\text{not changed}) \text{ otherwise} \end{cases}$
    For every pair of two units $j$ and $k$ where $j \neq k$, their weight $w_{jk}$ are updated according to
    Hebbian rule as follows:
        $w_{jk} = w_{jk} + \Delta w_{jk} = w_{jk} + y_j y_k$
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.

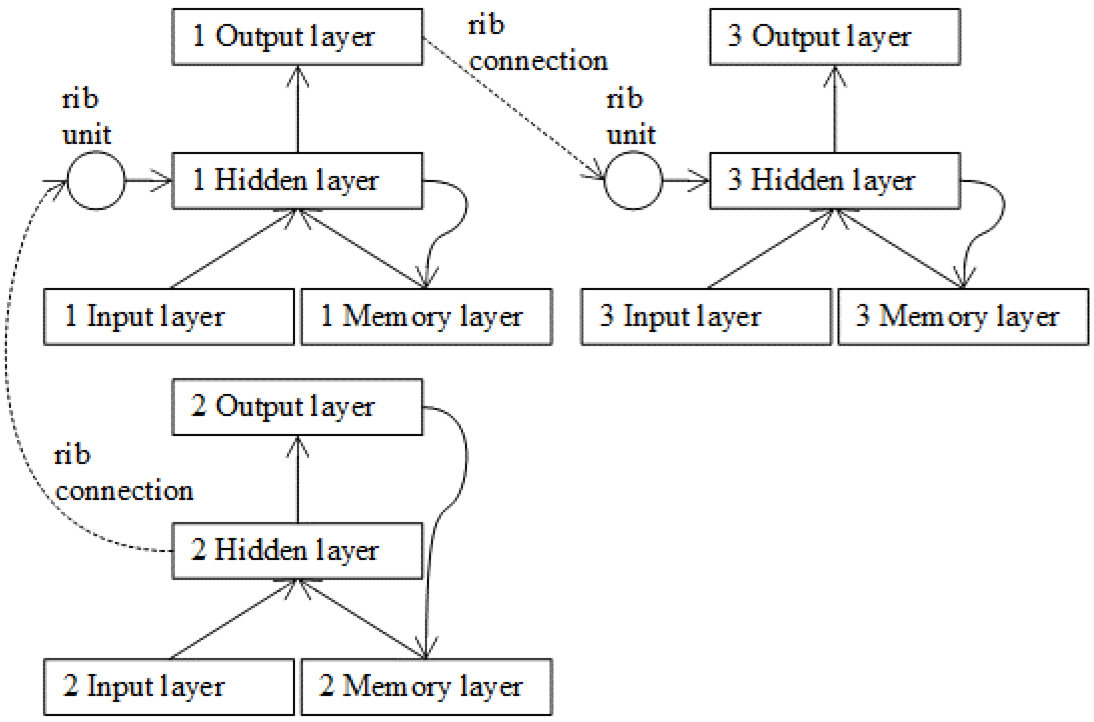

Figure 3.4. Fishbone neural network (FBNN)

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.

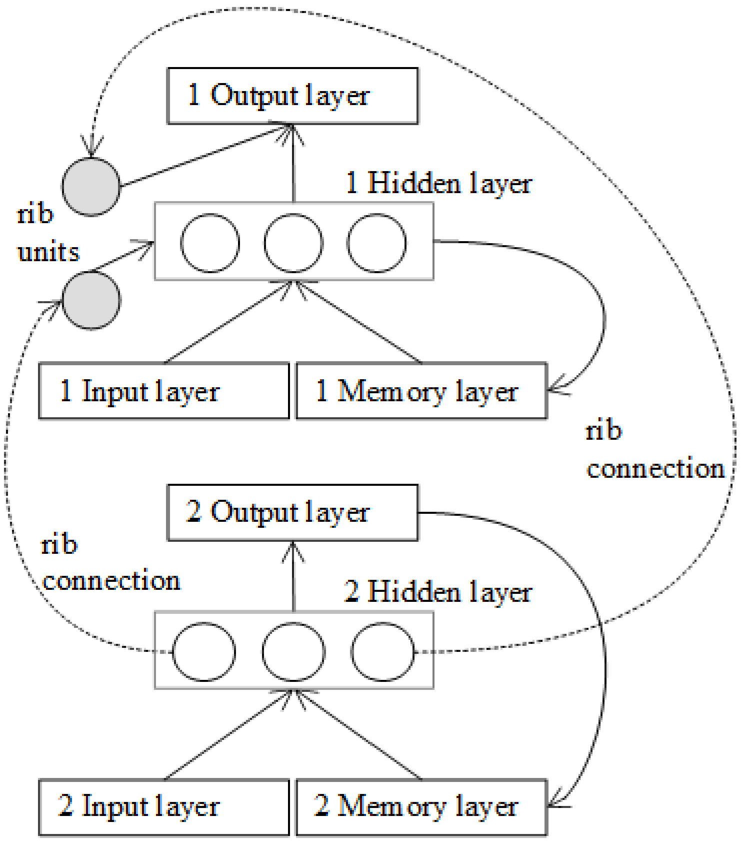

Figure 3.5. Two FBNNs connect together

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.

4. 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 $\boldsymbol{x}$, supervised learning algorithms improve weights and biases in order to make an approximation to the desired output function $v(\boldsymbol{x}) = \boldsymbol{v}$. However, in case that there is no desired outputs $\boldsymbol{v}$ as supervisors, learning algorithms must process only inputs $\boldsymbol{x}$, 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 $\boldsymbol{x} = (x_i)$ there is only one output unit *o* is valid, which is called activated output unit or *winner* (Kröse & Smagt, 1996, pp. 57-58).

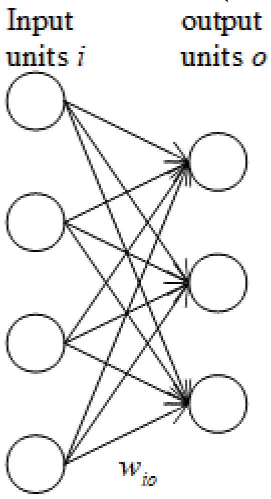

Figure 4.1. Simple network of competitive learning

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):

$$y_o = x_o = \sum_i w_{io} x_i = \boldsymbol{w}_o^T \boldsymbol{x} \tag{4.1}$$

Where $\boldsymbol{x} = (x_i) = (x_1, x_2,\ldots, x_n,\ldots)^T$ is input vector and $\boldsymbol{w}_o = (w_{1o}, w_{2o},\ldots, w_{no},\ldots)^T$ whereas y_o 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).

$$\forall o' \neq o, y_{o'} \leq y_o \tag{4.2}$$

After the winner was selected, its output is activated to be zero as $y_o = 1$ and other outputs of output units are deactivated to be zero as $y_{o'} = 0$ (Kröse & Smagt, 1996, p. 58).

$$\begin{gathered} y_o = 1 \\ \forall o' \neq o, y_{o'} = 0 \end{gathered} \tag{4.3}$$

Within dot product method, only weight vector $\boldsymbol{w}_o = (w_{1o}, w_{2o},\ldots, w_{no},\ldots)^T$ of the winner *o* is updated to be moved forward the input vector $\boldsymbol{x}$ and then normalized, as follows (Kröse & Smagt, 1996, p. 58):

$$\boldsymbol{w}_o = \frac{\boldsymbol{w}_o + \gamma(\boldsymbol{x} - \boldsymbol{w}_o)}{\|\boldsymbol{w}_o + \gamma(\boldsymbol{x} - \boldsymbol{w}_o)\|} \tag{4.4}$$

The denominator of equation above is used to normalize the winner weight vector $\boldsymbol{w}_o$ where notation ||.|| denotes Euclidean norm. Note, γ ($0 < \gamma \leq 1$) 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 $\boldsymbol{w}_o$ and the input vector $\boldsymbol{x}$ is minimum, which means that the winner *o* is the nearest to the input vector $\boldsymbol{x}$.

$$\forall o' \neq o, \|\boldsymbol{w}_{o'} - \boldsymbol{x}\| \geq \|\boldsymbol{w}_o - \boldsymbol{x}\| \tag{4.5}$$

After the winner was selected, its output is activated to be zero as $y_o = 1$ and other outputs of output units are deactivated to be zero as $y_{o'} = 0$.

$$y_o = 1$$
$$\forall o' \neq o, y_{o'} = 0$$

Like dot product method, only weight vector $\boldsymbol{w}_o = (w_{1o}, w_{2o},\ldots, w_{no},\ldots)^T$ of the winner *o* is updated to be moved forward the input vector $\boldsymbol{x}$ but such winner weight vector is often not normalized.

$$\boldsymbol{w}_o = \boldsymbol{w}_o + \gamma(\boldsymbol{x} - \boldsymbol{w}_o) \quad (4.6)$$

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

$$\boldsymbol{\varepsilon}(y_o) = \boldsymbol{\varepsilon}(\boldsymbol{w}_o) = \begin{pmatrix} \frac{1}{2}(w_{1o} - x_1)^2 \\ \frac{1}{2}(w_{2o} - x_2)^2 \\ \vdots \\ \frac{1}{2}(w_{no} - x_n)^2 \\ \vdots \end{pmatrix} \quad (4.7)$$

Gradient of the squared error of output unit *o* with regard to w_{io}, known as tangent vector of $\boldsymbol{\varepsilon}(\boldsymbol{w}_o)$, is:

$$\nabla\boldsymbol{\varepsilon}(\boldsymbol{w}_o) = \frac{d\boldsymbol{\varepsilon}(\boldsymbol{w}_o)}{dw_{io}} = \begin{pmatrix} x_1 - w_{1o} \\ x_2 - w_{2o} \\ \vdots \\ x_n - w_{no} \\ \vdots \end{pmatrix} = \boldsymbol{x} - \boldsymbol{w}_o \quad (4.8)$$

Note, $\frac{d\boldsymbol{\varepsilon}(\boldsymbol{w}_o)}{d\boldsymbol{w}_o}$ is Jacobian matrix but the equation above expresses tangent vector for easily understandable explanation.

$$\frac{d\boldsymbol{\varepsilon}(\boldsymbol{w}_o)}{d\boldsymbol{w}_o} = \begin{pmatrix} x_1 - w_{1o} & 0 & \cdots & 0 \\ x_2 - w_{2o} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ x_n - w_{no} & 0 & \cdots & 0 \end{pmatrix}$$

Obviously, the rule of updating winner weight vector $\boldsymbol{w}_o = \boldsymbol{w}_o + \gamma(\boldsymbol{x} - \boldsymbol{w}_o)$ 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 $½(w_{io} - x_i)^2$ rather than $½(w_{io}x_o - x_i)^2$. Exactly, the error is $½(w_{io}x_o - x_i)^2$ but x_o is assumed to be 1 as $x_o = y_o = 1$ because the output unit *o* is assumed to be the winner and hence, we have $½(w_{io}x_o - x_i)^2 = ½(w_{io}*1 - x_i)^2 = ½(w_{io} - x_i)^2$. 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 $\boldsymbol{x} = (x_1, x_2,\ldots, x_i,\ldots, x_m)$ is a vector in real vector space $\mathbb{R}^m$ and output vector $y = (y_1, y_2,\ldots, y_o,\ldots, y_n)$ is a vector in real vector space $\mathbb{R}^n$, there are some orderings which are defined in $\mathbb{R}^m$ and $\mathbb{R}^n$. 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:

$$g(o, o') = \exp(-\|y_o - y_{o'}\|^2) \quad (4.9)$$

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:

$$\boldsymbol{w}_o = \boldsymbol{w}_o + \gamma(\boldsymbol{x} - \boldsymbol{w}_o)$$

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

$$\boldsymbol{w}_{o'} = \boldsymbol{w}_{o'} + \gamma g(o, o')(\boldsymbol{x} - \boldsymbol{w}_{o'}), \forall o' \in nb(o) \quad (4.10)$$

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.

5. 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.

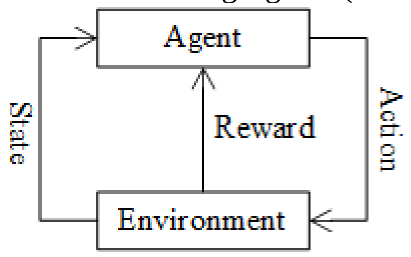

Figure 5.1. Overview of reinforcement learning

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.

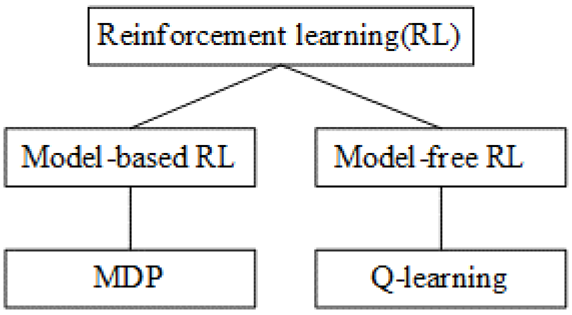

Figure 5.2. Roadmap of RL methodologies

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 s_t 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 a_t be the action at time point *t*.
- Let $P_a(s, s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$ be the transition probability at time point *t* from the current state $s_t = s$ to the next state $s_{t+1} = s'$ given action $a_t = a$. This transition probability is conditional probability. A set of all transition probabilities for all states given an action compose a transition probability matrix P_a. The transition probability implies that Markov property where the probability of next state *s*' depends only on current state *s*.
- Let $R_a(s, s')$ be the immediate reward that the environment issues immediately when the agent does the current action $a_t = a$ such that the current state $s_t = s$ is changed immediately to the next state $s_{t+1} = s'$. 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 $a = \pi(s)$. The essence of MDP is to train policy function $a = \pi(s)$ to be optimal, which in turn maximizes a so-called *value function* based on the immediate reward function $R_a(s, s')$ 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 $R_a(s, s')$ multiplied with discount factor α_t under the transition distribution $P_a(s, s')$. Therefore, *V*(s) is also called *discounted reward expectation*, which is determined from $s = s_{t_k}$ at some t_kth time point to infinity.

$$V(s = s_{t_k}) = E\left(\sum_{t=t_k}^{+\infty} \gamma_t R_{a_t}(s_t, s_{t+1})\right) = \sum_{t=t_k}^{+\infty} \gamma_t R_{a_t}(s_t, s_{t+1}) P_{a_t}(s_t, s_{t+1}) \qquad (5.1)$$

Where,

$$a_t = \pi(s_t)$$

Proof,

$$V(s = s_{t_k}) = E\left(\sum_{t=t_k}^{+\infty} \gamma_t R_{a_t}(s_t, s_{t+1})\right) = \sum_{t=t_k}^{+\infty} \gamma_t R_{a_t}(s_t, s_{t+1}) P\left(s_{t+1} \middle| s_t, s_{t-1}, \dots, s_{t_k}, a_t\right)$$

$$= \sum_{t=t_k}^{+\infty} \gamma_t R_{a_t}(s_t, s_{t+1}) P(s_{t+1} | s_t, a_t)$$

(Due to Markov property)

$$= \sum_{t=t_k}^{+\infty} \gamma_t R_{a_t}(s_t, s_{t+1}) P_{a_t}(s_t, s_{t+1}) \blacksquare$$

The discount factor α_t ($0 < \alpha_t \leq 1$) indicates that a reward $R_{a_t}(s_t, s_{t+1})$ can be delayed at time point t. The longer the delay is, the smaller discount factor is and so, only the first reward $R_{a_{t_k}}(s_t, s_{t+1})$ gains highest discount factor γ_{t_k}. If $\gamma_{t_k} = 1$ then, the first reward $R_{a_{t_k}}(s_t, s_{t+1})$ is immediate reward such that $R_{a_{t_k}}(s_t, s_{t+1})$ which is reserved. Discount factor should be inversely proportional to time point, for example $\alpha_t = 1 / (t+1)$. 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):

$$V(s) = \sum_{s'} P_{\pi(s)}(s, s') \left(R_{\pi(s)}(s, s') + \alpha V(s') \right) \quad (5.2)$$

Now value function is determined by a finite sum and so, it is called *discounted reward sum* in which $s \in S$, $a \in A$, and both S and A are finite sets. In first view, discount factor α 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 $R_{\pi(s)}(s, s')$ are reserved. Consequently, policy function $\pi(s)$ is updated as maximizer regarding value function as follows (Wikipedia, Markov decision process, 2004):

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left\{ \sum_{s'} P_a(s, s') \big(R_a(s, s') + \alpha V(s') \big) \right\} \quad (5.3)$$

An implementation of MDP learning is an iterative algorithm so that whenever the environment feeds back a next state s_{t+1} and gives back a reward $R_{a_t}(s_t, s_{t+1})$ for the agent's action a_t at the current state s_t (time point t), the iterative algorithm will update value and policy as follows:

Value update rule: $$V(s_t) = \sum_{s'} P_{a_t}(s_t, s') \left(R_{a_t}(s_t, s') + \alpha V(s') \right)$$ Policy update rule: $$\pi(s_t) = \underset{a}{\operatorname{argmax}} \left\{ \sum_{s'} P_a(s_t, s') \big(R_a(s_t, s') + \alpha V(s') \big) \right\}$$

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 $\pi(s)$. However, RL does not require mandatorily terminated conditions because it aims to adapt to the environment. Note that all values $V(s)$ and $R_a(s, s')$ for all s, s', and a are initialized by 0. Of course, the agent's action a_t at the current state s_t is based on the policy function $a_t = \pi(s_t)$ where s_t 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 $P_a(s, s')$ and reward function $R_a(s, s')$. With Q-learning, model-free RL broadens its applications, especially neural network learning. At time point t, the environment still gives back a reward R_t in model-free RL but such R_t is only a value which is not the function $R_a(s, s')$ in model-based RL. Given time point t, value function $V(s)$ in model-based RL is replaced by Q-value $Q(s_t, a_t)$ for model-free RL and such Q-value is learned as follows (Wikipedia, Q-learning, 2004):

$$Q(s_t, a_t) = Q(s_t, a_t) + \gamma\left(R_t + \alpha \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right) \tag{5.4}$$

Where γ ($0 < \gamma \leq 1$) is learning rate. The equation above is called Bellman equation. Therefore, whenever the environment feeds back a next state s_{t+1} and gives back a reward R_t for the agent's action a_t at the current state s_t (time point t). the iterative algorithm of Q-learning for model-free RL is described as follows:

Q-value update rule: $Q(s_t, a_t) = Q(s_t, a_t) + \gamma\left(R_t + \alpha \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right)$
Policy update rule: $\pi(s_t) = \underset{a}{\operatorname{argmax}}\, Q(s_t, a)$

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 $\pi(s)$. Of course, the agent's action a_t at the current state s_t is selected based on the policy function $a_t = \pi(s_t)$ where s_t 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 $v(\boldsymbol{x})$ where $\boldsymbol{x}$ is inputs, and the approximation can be represented by an estimation function $u(\boldsymbol{x})$. Essentially, the estimation function $u(\boldsymbol{x})$ 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 $u(\boldsymbol{x})$, increasing such order is obviously to increase the accuracy of $u(\boldsymbol{x})$ 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.

$$Q(s_t, a_t) = Q\big(x_k(t), y_k(t)\big)$$

Where $x_k(t) = s_t$ and $y_k(t) = a_t$ are input and output of unit k at time point t. Regarding NN, Q-value is Q-function of $x_k(t)$ and $y_k(t)$. 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 s_{t+1} and gives back a reward $R_k(t)$ for the agent's action $a_t = y_k(t)$ at the current state $x_k(t) = s_t$, the Q-function is updated as follows:

$$Q\big(x_k(t), y_k(t)\big) = R_k(t) + \alpha \max_k Q_0\big(x_k(t), y_k(t)\big)$$

Where α is discount factor and v_k is desired output. Note that index k in the maximization expression $\max_k Q_0\big(x_k(t), y_k(t)\big)$ indicates browsing units in the same layer of current unit. There is a question what $Q_0(x_k(t), y_k(t))$ is. Indeed, according to an invention of OpenAI (Choudhary, 2019), $Q_0(x_k(t), y_k(t))$ is the function $Q(x_k(t), y_k(t))$ 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 s_{t+1} in Q-learning. The following figure depicts the target network for Q-learning (Choudhary, 2019).

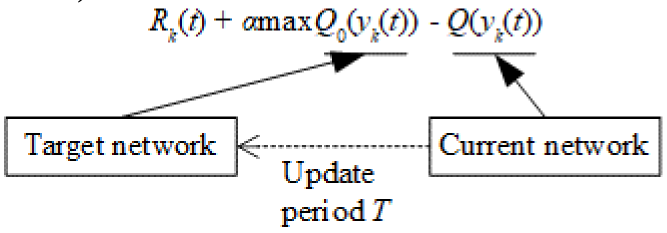

Figure 5.3. Target network for Q-learning

Because y_k is function of x_k due to activation function $y_k = f(x_k)$, Q-function in NN is considered as function of y_k as follows:

$$Q\big(y_k(t)\big) = R_k(t) + \alpha \max_k Q_0\big(y_k(t)\big)$$

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

$$\Delta Q\big(y_k(t)\big) = R_k(t) + \alpha \max_k Q_0\big(y_k(t)\big) - Q\big(y_k(t)\big)$$

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:

$$\Delta Q(y_k) = R_k + \alpha \max_k Q_0(y_k) - Q(y_k) \tag{5.5}$$

Note that the expression $\max_k Q_0(y_k)$ is constant with regard to y_k. Recall that index k in the maximization expression $\max_k Q_0(y_k)$ 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 $\Delta Q(y_k)$ above, only $Q(y_k)$ is function of y_k. The simplest way is to set Q-function as identity function $Q(y_k) = y_k$. Derivative of $\Delta Q(y_k)$ with regard to x_k is:

$$\frac{d\Delta Q(y_k)}{dx_k} = \frac{d\Delta Q(y_k)}{dy_k}\frac{dy_k}{dx_k} = -Q'(y_k)f'(x_k) \tag{5.6}$$

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

$$\varepsilon(y_o) = \frac{1}{2}\big(\Delta Q(y_o)\big)^2 = \frac{1}{2}\left(R_o + \alpha \max_o Q_0(y_o) - Q(y_o)\right)^2 \tag{5.7}$$

The squared error function $\varepsilon(y_h)$ of hidden unit h is the sum of output errors $\varepsilon(y_o)$ with regard to such set of output units, as follows:

$$\varepsilon(y_h) = \sum_o \varepsilon(y_o)$$

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

$$\Delta w_{jk} = \gamma y_j \delta_k$$
$$\Delta \theta_k = \gamma \delta_k$$

Where,

$$\delta_k = \begin{cases} \left(R_k + \alpha \max_k Q_0(y_k) - Q(y_k)\right) Q'(y_k) f'(x_k) \\ \text{for ouput unit} \\ \\ Q'(y_k) f'(x_k) \sum_l w_{kl} \delta_l \\ \text{for hidden unit} \end{cases} \quad (5.8)$$

Recall that:

$$\frac{d\Delta Q(y_k)}{dx_k} = -Q'(y_k) f'(x_k)$$

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

$$Q(y_o) = Q(y_o) + \gamma \left(R_o + \alpha \max_o Q_0(y_o) - Q(y_o)\right) \quad (5.9)$$

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 $\{\boldsymbol{x}^{(p)}, \boldsymbol{R}^{(p)}\}$ where input vector $\boldsymbol{x}^{(p)}$ is a set of states and $\boldsymbol{R}^{(p)}$ is a set of rewards of output units at p pattern. Agent's actions are outputs y_k from computations inside NN and next states s_{t+1} are represented by the target network.

6. 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.

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