Explaining and Reproducing "Performance Surfaces of a Single-Layer Perceptron" by John J. Shynk

1st Furkan Öztürk
Computer Engineering
Hacettepe University
Ankara, Turkey
furkan15ozturk@gmail.com

I. Inspecting: Abstract

In the abstract section of **Performance Surfaces of a Single-Layer Perceptron**, it is told that the perceptron learning algorithm may be viewed as a **steepest-descent method** whereby an instantaneous performance function is iteratively minimized.

The steepest-descent method or saddle-point method, also known as **gradient-descent method**, is a iterative process used to find the minimum point of a function. Although it is one of the simplest and known methods for minimizing a function, it has slow convergence rate.

The paper is about a widely-used perceptron algorithm that uses a appropriate performance function. This algorithm shows that the update term of the algorithm is the gradient of this function. The **update term** is the change made to the weight and biases. It is told that the **slope**, the **derivative** of this function, is equal to the update term.

It is told that an example is given of the corresponding performance surface based on Gauss assumptions. **Performance surface** is the surface full of points. Each point represents an error value, for a specific weight configuration. **Gauss assumptions**, also called **Gauss-Markov Theorem**, is used to make assumptions about regression coefficients. It helps us to check how well our data matches the assumptions. It is told there are infinite **stationary points**. A stationary point is a point of a function $f(x)$, where it is a point where the derivative of a function $f(x)$ is equal to 0.

It is told that two other related performance functions are examined. It is told that alternative perceptron algorithms can be derived from these performance functions.

II. Inspecting: Introduction

At the start of the Introduction section, the perceptron is defined as a linear combiner that quantizes its output to one of two discrete values. It is told that variety of different learning algorithms can be used adjusting the weights and the threshold (bias). It is said that the original perceptron convergence theorem converges and positions a hyperplane. This hyperplane is called the decision boundary and it separates two different classes. It is said that the perceptron with adaptive weights is also called **Adaline** (for *ada*ptive *l*inear *ne*uron)

It is said that the Least Mean Squares algorithm is similar to Rosenblatt's algorithm. It is said that Rosenblatt's algorithm is used primarily for a linear combiner without quantization, minimizing the difference between desired response and the linear output.

A definition is made for Multi-layer Perceptron and it is said that Multi-layer Perceptrons (Madelines) are feedforward neural networks with one or more layers of Adalines. A comparison between MLP (Multi-layer Perceptron) and SLP (Single-layer Perceptron) is made. While two or three layered perceptrons can represent complicated decision areas, SLP can only separate regions by a hyperplane.

System definitions are made for single-layer perceptron. The input signals are defined as $\{x_k\}$ and the adjustable weights are defined as $\{ w_k\}$. It is said that, the input signals $\{x_k\}$ and the adjustable weights $\{w_k\}$ are used in order to generate output signal y. The output signal y is processed by a hard limiter, resulting in the quantized binary output $y_q(\pm 1)$. This binary output is compared to the desired response d_q, which is also a binary signal, generating an error e_q that is used in a feedback strategy to adapt the weights $\{w_k\}$.

The method of adjusting the weights has been explained and it is said that an algorithm based on steepest descent is generally used to adjust the weights and to minimize a specific performance criterion. It is said that the input signals can be binary or can be drawn from a continuous distribution. In most cases the hard limiter is replaced by a smooth non linearity, such as sigmoid function. Below figure shows the sigmoid function:

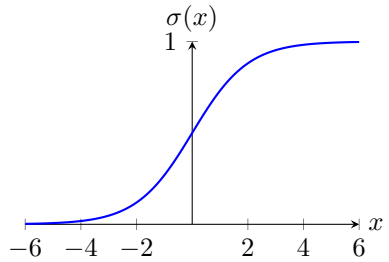

Fig. 1. Caption

An explanation is made for SLP and it is explained as single-layer perceptron essentially operates as a pattern classification device whereby the N-dimensional vector space, represented by the filter input signals, is partitioned by a hy-

perplane into two subspaces. Below figure shows an example of a hyperplane partitioning a N-dimensional vector space:

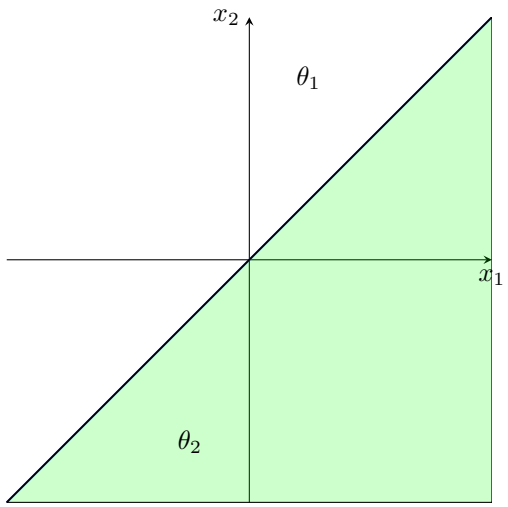

Fig. 2. Illustration of a hyperplane as a decision boundary in two-dimensional space, dividing θ_1 and θ_2.

It is said that for the case with two input signals ($N = 2$), the two-dimensional space is defined by $x_1 - x_2$ axes and the separating hyperplane is a straight line which partitions space into two half planes. Figure 2 shows two subspaces, θ_1 and θ_2. Blue line represents the hyperplane dividing the two-dimensional space into two half planes. An example is given with $b = 0$ and $w_1 = -w_2$. The partition becomes a line that goes through origin with a 45°. The location of the line is determined by the input signals $\{x_k\}$ and the desired response $\{d_q\}$.

In the paper it is said that the paper reviews certain aspects of the perceptron and describes a performance function for a widely used perceptron algorithm, which is viewed as a steepest-descent method. It is said that the performance function of the perceptron algorithm is not the usual mean-square-quantized error, but it has the same minimum points.

Next, the sections of the paper is introduced. Section II has the standard perceptron algorithm. It is about the perceptron algorithm and why it is similar to steepest-descent. In Section III, three performance functions are introduced. It is said that based on certain Gaussian assumptions, analytical expressions are derived for all three of these performance functions and their minimum points. Section IV has examples of the performance surfaces associated with these performance functions. Section V has the conclusions.

III. Inspecting: Perceptron Learning Algorithm

Found by Frank Rosenblatt, in 1958, was the early steps of developing Neural Networks. The perceptron is considered to be the simplest form of a neural network. The perceptron algorithm is basically a binary classifier. It consists adjustable weights and biases. It determines the weights based on input and outputs. If the output is different than expected results, weights are adjusted according to **perceptron convergence theorem**.

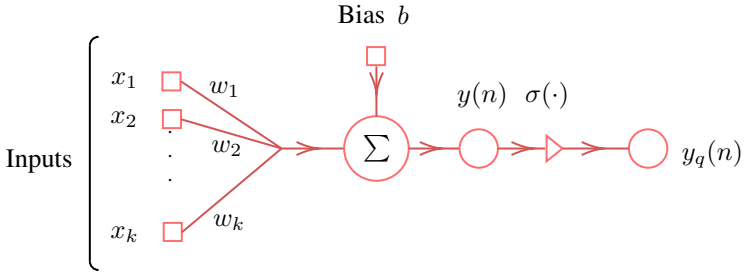

Fig. 3. Perceptron Model

The figure above demonstrates the signal flow of the perceptron model. The perceptron convergence theorem is given as:

$$W(n+1) = W(n) + 2\mu e_q(n)X(n) \tag{1}$$

where $W(n)$ and $X(n)$ are N-dimensional column vectors with components $\{w_k\}$ and $\{x_k\}$, respectively, and n is the discrete-time index. The positive step-size μ controls the convergence rate and steady-state properties of the algorithm. The error term $e_q(n)$ is the difference between $d_q(n)$ and the quantized output $y_q(n)$, where sgn(·) is the signum function. The error term $e_q(n)$ can be explained as:

$$e_q(n) = d_q(n) - y_q(n) = d_q(n) - \text{sgn}(y(n)) \tag{2}$$

The unquantized output $y(n)$ is explained as the inner product between weights and the input signals. The formula can be written as:

$$y(n) = W^T(n)X(N) \tag{3}$$

where the superscript T denotes transpose.

Following the paper, it is said that $y(n)$ is quantized as part of the perceptron structure, but, $d_q(n)$ is usually provided directly as a binary signal. It is said that $d_q(n)$ can be seen as the quantized version of some underlying process $d(n)$, according to:

$$d_q(n) = \text{sgn}(d(n) \tag{4}$$

It is said that this representation is completely general and it is a useful interpretation. It is stated that $d(n)$ is correlated with $X(n)$, if it is not correlated, than the perceptron signals have not been chosen properly. Since $d(n)$ is correlated with $X(n)$,it can be often represented as a function of the elements of $X(n)$. It is stated that one useful case would be considered is that $d(n)$ is a linear combination of the elements of $X(n)$:

$$d(n) = F^T(n)X(n) \tag{5}$$

where $F(n)$ is defined in a manner similar to $W(n)$, but with components $\{f_k(n)\}$. $F^T(n)$ corresponds to the line that is wanted to be fit. The perceptron algorithm tries to converge the weights so that it creates a line equal to the line of $F^T(n)$.

An example is given following the paragraph. It is said that a line passes through origin with a degree of 45°, $d(n) = x_2(n) - x_1(n)$ and $F = [-1\ 1]^T$. With this definition, the region above the line corresponds to $d_q(n) = +1$, and the region below the line corresponds to $d_q(n) = -1$. The line that is told can be shown as:

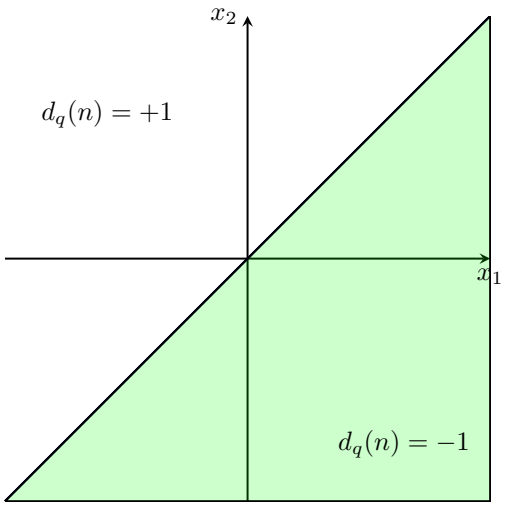

Fig. 4. Graph of how $d_q(n)$ responds

Now that the definition of perceptron and explaining perceptron convergence theorem is completed, how perceptron algorithm works can be explained.

1) Perceptron Convergence Theorem: Let's say that we have an $X(n)$ input vector, and this input is created by a signal that repeats every few seconds. The time-step will be represented as n. The bias for the input vector is represented as b and in order to add this value to the input vector, +1 value will be added. Thus, the input vectors size will be (m + 1)by-1 and will be defined as:

$$x_k(n) = [+1, x_1(n), x_2(n), ..., x_m(n)]^T \tag{6}$$

According to this input vector the weight vector will be defined as:

$$w_k(n) = [b, w_1(n), w_2(n), ..., w_m(n)]^T \tag{7}$$

Correspondingly, the hard limiter input will be defined as:

$$y(n) = \sum_{k=0}^{m} w_k(n)x_k(n) \tag{8}$$

$$= W^T(n)X(N) \tag{9}$$

Note that equation (8) has no bias value, since w_0 is used as bias. The next step is to calculate $d(n)$. As it is said before, the value of $d(n)$ can be calculated by using equation (5).

After calculating both $y(n)$ and $d(n)$ values, error value $e_q(n)$ must be calculated using equation (2). For that, signum function is need to be used:

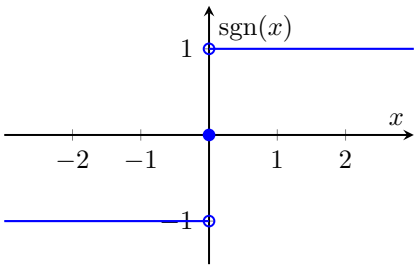

Fig. 5. Signum Function

This plot shows how signum function acts. Correspondingly to the input, the signum function gives output +1 or -1. After finding $y_q(n)$ and $d_q(n)$ based on the signum function, equation (2) can be applied.

Last step is to decide the step size value, μ:

$$0 < \mu \leq 1 \tag{10}$$

After all these steps, all that's left is to use equation (1) iteratively.

For example, the data that is trying to be classified has to be linearly separable. Below figure shows the data and the line that separates them:

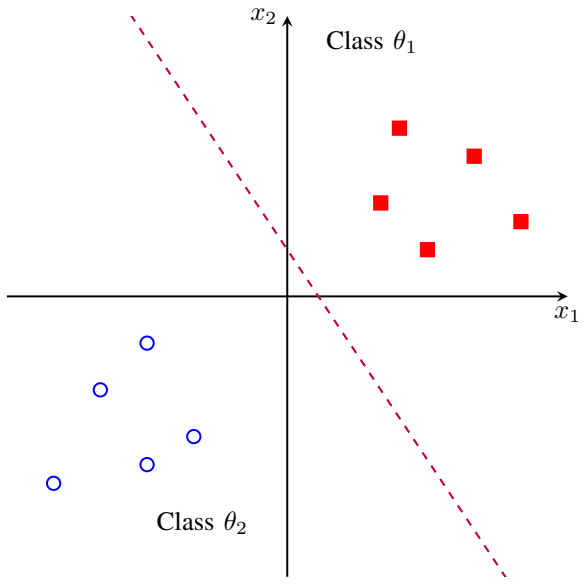

Fig. 6. A pair of linearly separable data

As can be seen above figure the data can be separated through a line. This would not happen in the case of mixed two classes of data. The line equation can be computed using **perceptron convergence theorem**.

Next, the algorithm in equation (1) is compared to LMS (Least-Mean Squares) algorithm, and it is said that they are similar except that the filter output is quantized to generate $y_q(n)$.

The LMS algorithm is developed by Widrow and Hopf in 1960, and was the first linear adaptive-filtering algorithm. This algorithm can solve problems such as prediction and

communication-channel equalization. It is used throughout many fields such as signal and image processing, digital filters, prediction models, and machine learning algorithms. The common feature between LMS and Perceptron algorithm is that they both involve use of a linear combiner.

It is said that the LMS algorithm is based on the method of steepest descent and it attempts to minimize an instantaneous estimate of the mean-squared error. This estimate is explained as $e^2(n)$ where $e(n) = d(n) - y(n)$.

Steepest Descent algorithm is explained as:

$$W(n+1) = W(n) + \mu\nabla(n) \tag{11}$$

where $\nabla(n)$ is an instantaneous estimate of the gradient ∇ of the performance function. LMS algorithm is explained as:

$$\nabla(n) = -2e(n)X(n) \tag{12}$$

This formula comes from differentiating $e^2(n)$ with respect to $W(n)$. Substituting equation (12) to equation (11) gives a similar result to equation (1). The main difference between LMS and Perceptron is that the LMS algorithm is a linear function of the signals, whereas the perceptron algorithm involves a nonlinear function via the hard limiter.

IV. Inspecting: Perceptron Performance Functions

The performance functions are denoted by ξ_m, their gradients by ∇_m, and the minimum points by W_m, all of which are independent of n.

A. Performance function based on $|y(n)|$

An instantaneous performance function is given as:

$$\xi_1 = 2|y(n)| - 2d_q(n)y(n) \tag{13}$$

Differentiating this equation with respect to $W(n)$:
Step 1: Take derivative of $|2y(n)|$

$$\frac{\partial}{\partial W(n)} 2|W^T(n)X(n)| \tag{14}$$

Case 1: When

$$W^T(n)X(n) \geq 0, |W^T(n)X(n)| = W^T(n)X(n) \tag{15}$$

Case 2: When

$$W^T(n)X(n) < 0, |W^T(n)X(n)| = -W^T(n)X(n) \tag{16}$$

The rule is that:

$$\frac{\partial}{\partial W(n)}(W^T(n)X(n)) = X(n) \tag{17}$$

Case 1: $W^T(n)X(n) \geq 0$

$$\frac{\partial}{\partial W(n)}(2W^T(n)X(n)) = 2X(n) \tag{18}$$

Case 2: $W^T(n)X(n) < 0$

$$\frac{\partial}{\partial W(n)}(-2W^T(n)X(n)) = -2X(n) \tag{19}$$

Combining the results:

$$\frac{\partial}{\partial W(n)}(2|y(n)|) = \begin{cases} 2X(n) & \text{if } W^T(n)X(n) \geq 0 \\ -2X(n) & \text{if } W^T(n)X(n) < 0 \end{cases} \tag{20}$$

Final answer:

$$\frac{\partial}{\partial W(n)}(|2y(n)|) = 2\text{sgn}(W^T(n)X(n))X(n) \tag{21}$$

Step 2: Take derivative of $-2d_q(n)y(n)$
Substitute $y(n)$:

$$-2d_q(n)W^T(n)X(n) \tag{22}$$

Use the rule from equation (17):

$$\frac{\partial}{\partial W(n)}(-2d_q(n)W^T(n)X(n)) = -2d_q(n)X(n) \tag{23}$$

Final Step: Merge derivations:

$$\begin{aligned} \nabla_1 &= 2\text{sgn}(W^T(n)X(n))X(n) - 2d_q(n)X(n) \\ &= -2e_q(n)X(n) \end{aligned} \tag{24}$$

With this process the equation in the paper can be derived. It is said that by substituting this result into (11), the algorithm in (1) is obtained. It is stated that if the algorithm is seen as the steepest-descent method, then equation (13) is the instantaneous performance function which the algorithm attempts to minimize. Later on, it is stated that by observing $|y(n)| = \text{sgn}(y(n))y(n)$, equation (13) can be written as:

$$\xi_1(n) = 2y_q(n)y(n) - 2d_q(n)y(n) = -2eq(n)y(n) \tag{25}$$

Note that if the derivative is taken with respect to $W(n)$, the equation (**??**) is obtained.

B. Performance function Based on $e_q^2(n)$

Given performance function is:

$$\xi_2(n) = e_q^2(n) \tag{26}$$

The difficulty of this performance function is explained as its derivative involves an impulse function. The derivative of the sign function is $2\delta(y(n))$. δ is explained as Dirac delta function. As a result, derivative of this performance function is not useful in a steepest descent algorithm.

Unlike the first performance function, this implementation replaces sgn$y(n)$ with a sigmoidal function of the form

$$g(y(n)) = 2f(y(n)) - 1 \tag{27}$$

where

$$f(y(n)) = (1 + e^{-\alpha y(n)})^-1 \tag{28}$$

α is explained as a positive scalar that is used to modify the shape (slope) of $g(y(n))$; as $\alpha \longrightarrow \infty, g(y(n)) \longrightarrow sgn(y(n))$.

Taking derivative of $g(y(n))$:

1. Differentiating $g(y(n)) = 2f(y(n)) - 1$:

$$g'(y(n)) = 2\frac{d}{dy(n)}f(y(n)) \tag{29}$$

2. Using the chain rule, the derivative of $f(y(n))$ is:

$$\frac{d}{dy(n)}f(y(n)) = \alpha f(y(n))(1 - f(y(n))) \tag{30}$$

3. Substitute this result into $g'(y(n))$:

$$g'(y(n)) = 2\alpha f(y(n))(1 - f(y(n))) \tag{31}$$

Therefore, the derivative of $g(y(n))$, with respect to $y(n)$ is:

$$g'(y(n)) = 2\alpha f(y(n))[1 - f(y(n))] \tag{32}$$

Following two plots, Figs. 7 and 8 show that $g(y(n))$ and $g'(y(n))$ values, respectively, for three different values α. Gradient of the performance function:

$$\nabla_2(n) = -4\alpha e_q(n) f(y(n))[1 - f(y(n))]X(n) \tag{33}$$

and the corresponding perceptron algorithm is

$$W(n+1) = W(n) + 4\mu\alpha e_q(n) f(y(n))[1 - f(y(n))]X(n) \tag{34}$$

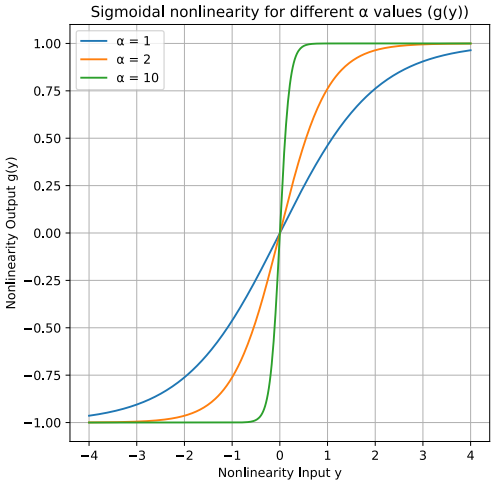

Fig. 7. Sigmoidal nonlinearity for three values of α

C. Performance function Based on $y^2(n)$

The performance function is given as:
The performance function is given by:

$$\xi_3(n) = [y(n) - d_q(n)]^2 \tag{35}$$

To find the gradient of this function with respect to $y(n)$, we proceed as follows:

1. Differentiating $\xi_3(n)$ with respect to $y(n)$:

$$\frac{d\xi_3(n)}{dy(n)} = 2[y(n) - d_q(n)] \cdot \frac{d}{dy(n)}(y(n) - d_q(n)) \tag{36}$$

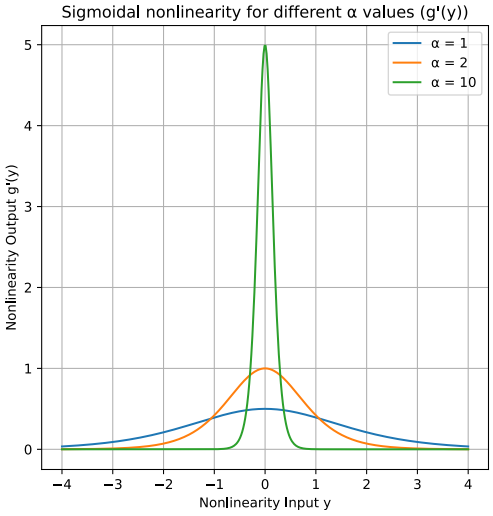

Fig. 8. Derivative of the sigmoidal nonlinearity for three values of α

2. Simplifying:

$$\frac{d\xi_3(n)}{dy(n)} = 2[y(n) - d_q(n)] \tag{37}$$

3. Since $y(n) = W^T X(n)$, as shown in equation (3), taking its gradient with respect to W:

$$\nabla_3(n) = \frac{\partial\xi_3(n)}{\partial W} = 2[y(n) - d_q(n)] \cdot X(n) \tag{38}$$

4. For minimization, negative gradient is used:

$$\nabla_3(n) = -2[d_q(n) - y(n)]X(n) \tag{39}$$

Equation (39) is similar to (24). This way, the corresponding algorithm is derived:

$$W(n+1) = W(n) + 2\mu[d_q(n) - y(n)]X(n) \tag{40}$$

V. Inspecting: Examples of the Performance Surfaces

A. Performance Surfaces

I inspected the surface plots and recreated them using Python and Matplotlib - Numpy libraries. With a given interval, same results can be achieved as can be seen in Fig. 9, 10 and 11.

The data that is being used has these following features:
$N = 2$
$R = I$(Identity matrix)

and the F vector to determine $d_q(n)$ is:
$F = [-1 - 1]^T$

Figures 9, 10 and 11 show the surfaces for ξ_1, ξ_2 and ξ_3. I recreated the same results in the paper. The minimum points for ξ_1 and ξ_2 starts from the origin and extending out such that $w_1 = w_2$ and $w_2 < 0$, as predicted in the paper. This is because the chosen elements for F vector.

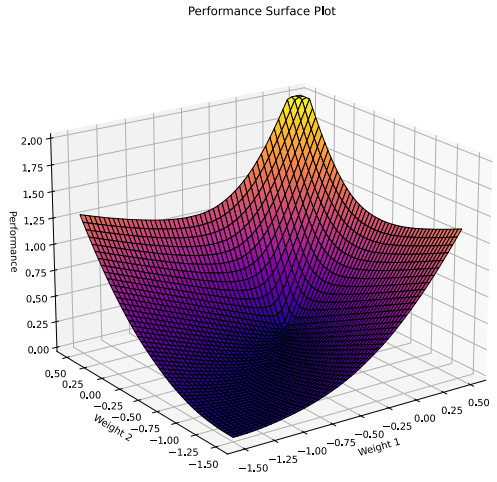

Fig. 9. Performance surface of ξ_1 truncated at 2.0

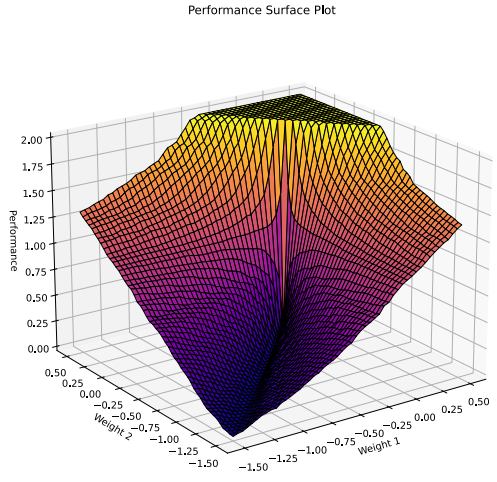

Fig. 10. Performance surface of ξ_2 truncated at 2.0

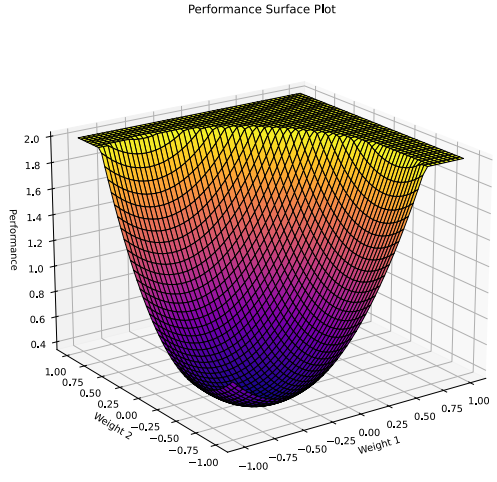

Fig. 11. Performance surface of ξ_3 truncated at 2.0

B. Weight Trajectories

Variables that are used calculating weight trajectories are:

The F vector to determine $d_q(n)$ is $F = [-1 - 1]^T$. The μ, learning rate is $= 0.01$. Sigmoidal nonlinearity with $\alpha = 1$. Independent computer runs = 25. Iterations $= 1000$.

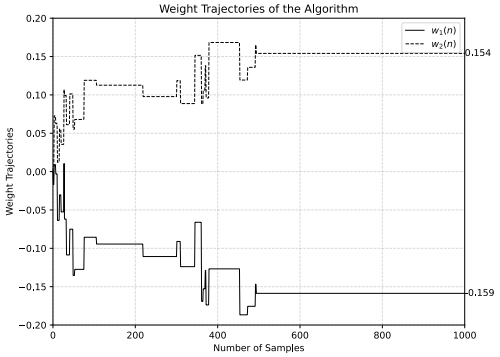

Fig. 12. Weight trajectories of the algorithm in (1)

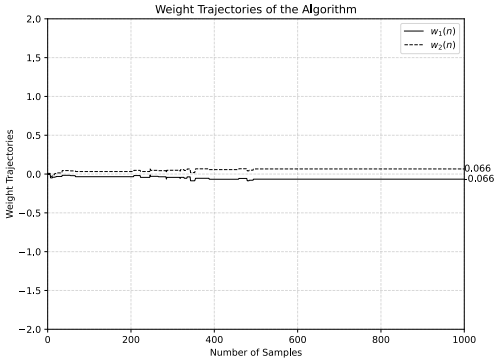

Fig. 13. Weight trajectories of the algorithm in (34)

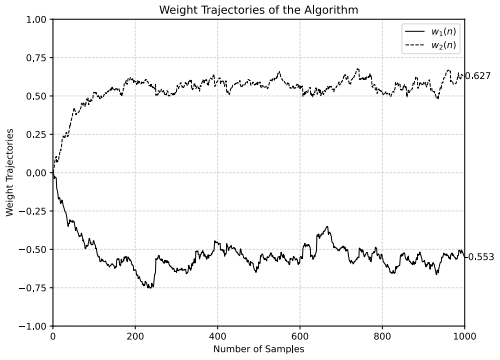

Fig. 14. Weight trajectories of the algorithm in (40)

Comparing the figure 12 with the paper, we can see that, resulting weight values are almost the same, although the trajectories are quite different. The figure 13 is quite different the one that is in paper, different results and different trajectories. The third figure, 14 shows quite resemblance with the one that

is in the paper, similar trajectories, similar results. From these trajectory plots, it can be easily seen that $w_1(n) \approx -w_2(n)$. The paper says that the weights in figures 12 and 13, in sense that they are not level as in 14. Although, from the plots I created, we can see that no change is made after approximately 500 iterations in figure 12. Figure 13 shows the same results too, after some point, the weight values are converged and has no change. In figure 14, we can see that the change is continuous, but, the weight values are level.

VI. Comments on "Performance Surfaces of a Single Layer Perceptron"

The codes below show how I implemented the perceptron update mechanism:

For performance function ξ_1:

```
self.weights[j] += 2 * self.learning_rate
                     * e_q_n
                     * self.training_data[i][j]
```

For performance function ξ_2:

```
self.weights[j] += 4 * self.learning_rate
                     * self.alpha
                     * e_q_n
                     * self.f(y_n)
                     * (1 - self.f(y_n))
                     * self.training_data[i][j]
```

For performance function ξ_3:

```
self.weights[j] += 2 * self.learning_rate
                     * (d_q_n - y_n)
                     * self.training_data[i][j]
```

These codes are in a loop where the loop iterates over jointly Gaussian dataset.

Even though, for the first two algorithms I did not get similar results, I did get a similar result when using the third algorithm. Each of these algorithms converged and get a nice result compared to the line that is wanted to be fit, which is F.

I tried different sample counts, all of the algorithms seem to work even with only two data points, given 10 or more epochs. I used only 1 epoch in my algorithm since I have 1000 data points generated by me, randomly, using Numpy library.

Numpy library generates data with a given mean. In this papers case, with a zero mean. Although, Numpy did not get as close as to 0 as I expected. So, what I did was to find the mean, and subtract it from each data point. This, I normalized the data, and the mean of the dataset was approximately 0, as it is said in the paper. The code below generates the dataset:

```
data =
np.random.multivariate_normal([mean]*N,
                                    R,
                                    M)
```

Some examples from the dataset:

```
   Signal 1                Signal 2
 3.769561962945e-01,   -9.759623443032e-01
-7.582614620577e-01,   -9.372960970987e-01
-4.774588654725e-01,   -4.349623916005e-01
```

Please visit this Github Repository of the implementation for this paper for more.