CSC 470: Introduction to Neural Networks

Module 6

Least Mean Square Algorithm

Objectives:

1. Understand what is meant by an "adaptive filter".

2. Understand the difference between spacial inputs vs. temporal inputs.

3. Be familiar with the concept of gradient descent and how it is used to solve an optimization problem such as finding an optimal weight vector value.

4. Be familiar with the concepts behind the following unconstrained optimization solution methods and how they compare to each other:
    a. Steepest descent method
    b. Newton's method
    c. Gauss-Newton method

5. Be introduced to the concept of a Wiener filter.

6. Understand the terms, *ensemble* and *ergodic*.

7. Be familiar with the basic concept of the least-mean-square (LMS) algorithm, and the advantage it has over the Wiener filter.

Textbook:

Haykin, Chapter 3, Sections 3.1-3.5 (other sections are informative, but optional)

(Again, do the best you can with the math; I've tried to explain certain things in more detail in the lecture, but please email me if you have any questions.)

Assignment:

No assignment for this module.

Quiz:

Complete the quiz for Module 6.

6.1 Adaptive Filters and The Least-Mean-Square (LMS) Algorithm

The **least-mean-square**, or LMS, algorithm is one of the most commonly used algorithms in machine learning. It was developed by Bernard Widrow and Ted Hoff at Stanford University in 1960 [1], inspired by the McCulloch-Pitts neuronal model of 1943 [2]. Its use was initially intended in the area of communications signal processing as an **adaptive filter** to remove noise or other undesired parts of a signal. By "adaptive", we mean a filter that has variable parameters that can be adjusted, allowing the filter great flexibility in how it behaves. The name of the algorithm is derived from the value it is intended to calculate. Given a series of signal inputs, the error signal — the difference between the desired output and the actual output, is computed for each input. The mean error signal is then calculated, and the goal of the LMS algorithm is to find a set of parameters that minimizes the mean error signal. Haykin mentions three advantages of the LMS algorithm, namely:

1. It is computationally cheap, relatively speaking, as it can be computed in linear time.
2. It is easy to code.
3. It is highly robust with respect to external disturbances, making it very reliable.

Before introducing the LMS algorithm we need to talk about a generic model for adaptive filters such as the LMS algorithm. We can use a signal processing diagram that is very similar to the diagram for the perceptron. Figure 6.1 shows an example of such a diagram.

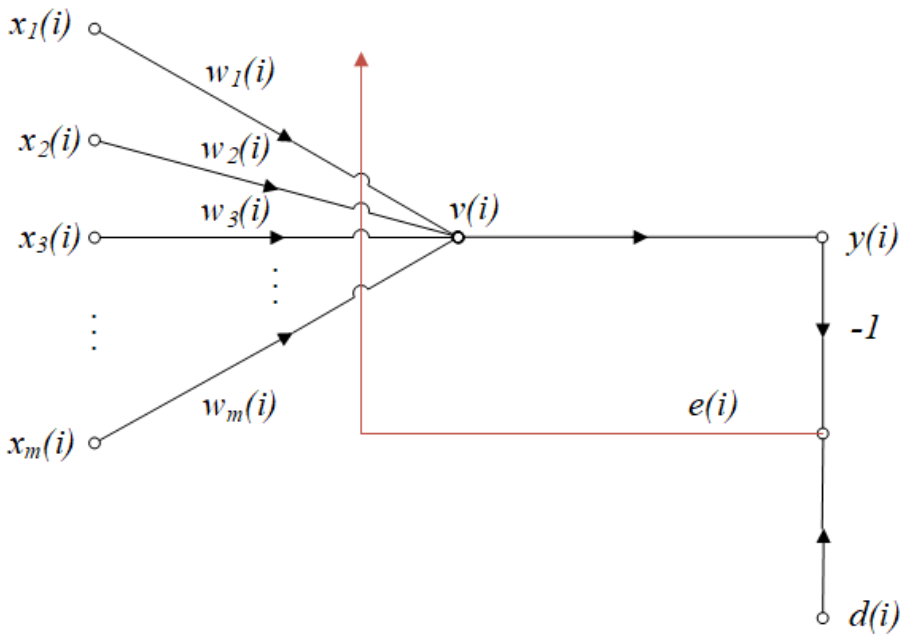

Figure 6.1. Signal processing diagram for an adaptive filter.

As you can see from the figure, the diagram is almost exactly the same as the signal processing diagram for the perceptron. There is an input vector, $\mathbf{x}$, consisting of inputs $x_1 \ldots x_m$, a corresponding synaptic weight vector, $\mathbf{w}$, consisting of weights $w_1 \ldots w_m$, an induced local field, v, and an output, y. What is different is that the desired output, d, is fed to the model externally, and the error signal, e, computes the difference between the desired output and the actual output, and the weights are adjusted by this error value. The 'i' in parentheses indicates the diagram only represents a discrete instant in time. Also, there is no hard limiter or activation function, meaning the output will not be restricted to two values.

We can use the perceptron we have discussed previously to model this filter. To train the model we need a training set that contains **both** the inputs and the desired output, since the desired output is needed to calculate the error signal. Notationally, the training set can be represented as:

$$T: \{\mathbf{x}(i), d(i); i = 1, 2, 3, ..., n, ...\}$$

where:

$$\mathbf{x}(i) = [x_1(i), x_2(i), x_3(i), ..., x_M(i)]^T$$

$d(i)$ is a scalar value

i represents an instant in time

M is the dimensionality of the input space

The source of the input values in $\mathbf{x}(i)$ can arise either **spacially** or **temporally**. The origins of the inputs are fundamentally different for each source, and each source defines the nature of the variable, i, in distinct ways:

- Inputs taken from a **spacial** origin are all taken from *different locations* in the input space at the *same time index*, i. The value of i is the same for each input value. This is often referred to as a **snapshot** of the data.

- Inputs taken from a **temporal** origin are all taken from the *same locations* in the input space, but at *different time indexes*. In this case the value of i changes. It is assumed that the interval of time between two inputs is the same; i.e., is uniform.

We need to make a few more assumptions to complete the model. Specifically:

1. The neuron's synaptic weights will initially be set arbitrarily; i.e., we will assign the initial weights as we see fit.

2. The adjustments made to the synaptic weights by the error signal will happen on a *continuous* basis. In other words, there will be a time factor involved.

3. All computations of the adjustments made to the synaptic weights must be completed within an interval whose length is equal to one sampling period. Any longer than this and the neuron will be out of sync with the inputs.

The **filtering process** of the model consists of the computation of two values:

1. the output, $y(i)$
2. the error signal, $e(i)$

The **adaptive process** of the model is the adjustment of the synaptic weights influenced by the error signal. The involvement of the error signal is crucial, since that is what makes the process adaptive, as the adjustments are dertermined by the value of the error signal. Together, the filtering and adaptive processes create a feedback loop that acts on the neuron through the continuous adjustments being made on the synaptic weights by the error signal.

6.2 The Unconstrained Optimization Problem

In the last module we talked briefly about the strategy of homing in on the optimal value for the weight parameter vector, $\mathbf{w}$, using a cost function, where the cost is the sum of the squares of the error signal over N trials:

$$E_0(\mathbf{w}) = \sum_{i=1}^{N} \varepsilon_i^2(\mathbf{w})$$

Essentially, what we want to do is find an optimal weight parameter vector, $\mathbf{w}^*$, whose cost is less than or equal to the cost of all other values of $\mathbf{w}$. The inequality that describes this would be:

$$E(\mathbf{w}^*) \leq E(\mathbf{w})$$

We must consider the inequality, "less than or equal to", rather than simply "less than", since it is possible for the current value of $\mathbf{w}$ to be the optimal value.

It is important that the cost function be *continuously differentiable*. A continuous function is simply a function that has no gaps; i.e., for every value of x there will be a corresponding value of y. "Continuously differentiable" means that not only must the derivative of the cost function exist, the derivative must itself be a continuous function.

The optimal value for $\mathbf{w}$ will be the value that results in the minimum cost. Thus, if the cost function were to be graphed with the cost as a function of $\mathbf{w}$, we would want the value for $\mathbf{w}$ that is a **local minimum**. Without going into too much math, we can use an iterative process that gradually adjusts the value of $\mathbf{w}$ such that the cost function continually decreases until it reaches a local minimum. This iterative process is called **gradient descent**, or more specifically, **iterative descent**, which is a very commonly employed optimization algorithm. The gradient is the derivative of a function of several variables (like a vector), and what we are actually interested in is the downward slope of this gradient. The slope is the change in all the weights in $\mathbf{w}$ with respect to the change in cost. The iterative gradient descent algorithm can be summarized as follows:

1. Start with an initial guess denoted by $\mathbf{w}(0)$.
2. Generate a sequence of weight vectors $\mathbf{w}(1)$, $\mathbf{w}(2)$, $\mathbf{w}(3)$, ..., such that the cost function $E(\mathbf{w})$ is reduced at each iteration of the algorithm; i.e.:

   $$E(\mathbf{w}(n+1)) < E(\mathbf{w}(n))$$

3. where $\mathbf{w}(n)$ is the old value of the weight vector and $\mathbf{w}(n+1)$ is the updated value of $\mathbf{w}$.

Given enough trials, the algorithm will hopefully converge onto the optimal solution, $\mathbf{w}^*$. For the optimal solution the following must hold true:

$$\nabla E(\mathbf{w}^*) = 0$$

The upside-down Greek delta character is pronounced "del", and represents the gradient.

6.3 Three Methods for Solving Unconstrained Optimization Using Iterative Descent

Haykin describes three different strategies for solving the unconstrained optimization problem utilizing iterative descent. These strategies are summarized below.

Method of Steepest Descent

If we imagine the cost function of the weight vector, $E(\mathbf{w})$, plotted on a graph, the line or curve representing the function will have a number of slopes depending on the shape of the curve. If we are trying to find inputs that correspond to a local minimum of the cost function, to give us $E(\mathbf{w}^*)$, one straightforward approach would be to descend the curve as quickly as possible. The fastest way to do this would be to simply travel is exactly the opposite direction of the curve's slope. Let us define a gradient vector evaluated on $\mathbf{w}$ as follows:

$$\mathbf{g} = \nabla E(\mathbf{w})$$

We can then define the steepest descent algorithm as follows:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta\mathbf{g}(n)$$

In this formula, n indicates the current sample in the iteration and $n+1$ indicates the next sample. The Greek letter, eta, is used to represent what is known as a **step size parameter** or **learning rate parameter**. This parameter is a positive value that modifies the gradient vector, $\mathbf{g}$, and this modified gradient vector value is subtracted from the weight vector to produce a new weight vector. As the algorithm iterates from n to $n+1$, the value of the correction to be applied to the weight vector is calculated as follows:

$$\begin{aligned}\Delta\mathbf{w}(n) &= \mathbf{w}(n+1) - \mathbf{w}(n) \\ &= -\eta\mathbf{g}(n)\end{aligned}$$

Thus, each weight in the weight vector is adjusted by the value, $-\eta\mathbf{g}(n)$.

The convergence of the weight vector toward the value that will minimize the cost function is affected significantly by the value of the learning rate parameter, η:

- Small values for η result in a smooth trajectory of descent.
- Large value for η result in a rough, oscillatory trajectory. There will still be descent, overall, but the descent will be somewhat inefficient.
- When η exceeds a certain threshold value, which will be dependent on the gradient function, the algorithm will become unstable, and instead of converging on a local minimum it will diverge away from the minimum.

Newton's Method

A drawback to the steepest descent method we just covered is the tendency of the algorithm to sometimes zigzag as it descends. An alternate approach is **Newtons' method**, which has an advantage over the steepest descent method in that it tends to descend quickly without the potentially destabilizing zigzag behavior. This comes at a cost, however. Newton's method is much more computationally complex than the steepest descent method. As Haykin illustrates in Chapter 3, the **gradient vector** of a cost function is a vector consisting of partial derivatives of the cost function with respect to each individual weight value in the weight vector:

$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_M} \right]^T$$

Newton's method attempts to optimize the weight vector to minimize the quadratic approximation of the cost function using a second-order Taylor series expansion, which is a common method of approximating functions. The added computational complexity of this method arises due to the need to generate for each iteration of the descent algorithm a matrix consisting of second partial derivatives with respect to *two* of the individual weight values of the weight vector. This matrix is known as a **Hessian matrix** (or more commonly, just **Hessian**). I don't expect you to be able to calculate Hessians, but I do want to at least introduce you to the terminology. Haykin shows an example of what the Hessian for Newton's method looks like on page 96.

In short, the convergence behavior of Newton's method is superior to that of the steepest descent method, but the added computational complexity may make using Newton's method impractical.

Gauss-Newton Method

As a compromise between the steepest descent method and Newton's method, we can use the **Gauss-Newton method**. For this method the cost function is defined in terms of the sum of the squares of the error values:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} e^2(i)$$

It should be clear that the value of the error signal at any given time will be dependent on the value of the weight vector, $\mathbf{w}$. The Gauss-Newton method linearizes this dependence as described by the following equation:

$$e'(i, \mathbf{w}) = e(i) + \left[\frac{\partial e(i)}{\partial \mathbf{w}} \right]^T_{\mathbf{w}=\mathbf{w}(n)} \times (\mathbf{w} - \mathbf{w}(n)), \quad i = 1, 2, ..., n$$

As you can see, the Gauss-Newton method still has some computational complexity, but not as much as Newton's method. It still involves the computation of a matrix of derivatives of the cost function, but instead of needing to compute a matrix of second-order partial derivatives it only needs to compute a matrix of first-order derivatives. (This special type of matrix is known as a **Jacobian**.)

6.4 Forward Problems vs. Inverse Problems

Before continuing I want to take a brief aside to introduce a couple of new terms that you may see from time to time, and that help to put into perspective what it is we're trying to accomplish with neural networks and machine learning in general. We have talked a lot about using a perceptron as a model for certain types of problems. Given a dataset, we train the perceptron to accurately predict the desired output from a set of inputs. Once we have the perceptron trained, we can then use it to predict the outputs for input vectors the perceptron has never seen before. Using a model to predict new information about an environment is known as a **forward problem**, since we are trying to go from *cause* (the input vector) to *effect* (the output). When we use data to train a neural network in order to find a weight vector that will produce the desired output from a set of inputs, we are working in the opposite direction, going from effect to cause. This type of problem is known as an **inverse problem**, and is characteristic of the problems we are trying to solve with machine learning and neural networks. We have data, and we are trying to find patterns in the data in order to build a model that accurately represents the problem at hand.

6.5 The Wiener Filter

Data from audio signals, video signals, images, as well as from other sources, are often not clean. It is typical for there to be noise in the data in the form of static, graininess, blurriness, and obstructions. Removing or at least reducing the noise in these types of signals is a very common problem. The goal is to take the signal data and "filter out" the noise so that we are left with a sharper image, clearer audio, less grainy video, etc. The **Wiener filter** is a very common solution to this type of problem. Wiener theory, developed by Norbert Wiener in the 1940's, describes a class of linear least squared filters that can be used in a broad array of applications. Reportedly, Wiener initially developed the theory to analyze radar data in order to predict positions of German bombers during World War II.

Wiener filters can be either linear or non-linear depending on how they are implemented. They can also be adaptive, even though Wiener theory originally assumed stationary signal processes. A **stationary signal** is one where the time period, frequency, and spectral contents of the signal remain constant over time. Probably the best example of a stationary signal is white noise, which in itself is interesting because while it is often desirable to filter out white noise to sharpen a signal, there are also many applications for generating or enhancing white noise in a signal, such as masking tinnitus or as an aid to sleeping. By contrast, a **non-stationary signal** is one where the time period, frequency, and/or the spectral contents are dynamic and change with time. Most of the signals we try to enhance, such as image data and speech, are non-stationary.

Haykin derives a new formula for the least squares problem in section 3.4 of the text. The math in the derivation is rather involved, so I will just summarize the final formula here along with the implication that results. The formula, as derived by Haykin, is:

$$\mathbf{w}(n+1) = \mathbf{X}^{+}(n)\mathbf{d}(n)$$

The term, $\mathbf{X}^{+}(n)$, is the **pseudoinverse** of the matrix, $\mathbf{X}(n)$, which is the matrix of input vectors to the model:

$$\mathbf{X}(n) = [\mathbf{x}(1), \mathbf{x}(2), ..., \mathbf{x}(n)]^{T}$$

In the linear algebra primer I presented back in Module 2 I described the matrix inverse operation, which is a matrix that when multiplied by another matrix produces an identity matrix of the same dimensions. The inverse operation only works for square matrices,

however, and in the case of our filter model, $\mathbf{X}(n)$ has dimensions $n \times M$. The pseudoinverse can be computed using singular value decomposition (SVD).

The implication of this is that if we are observing a system for a time interval of length n, or a number of observations equal to n, the weight parameters for the next observation, at $n + 1$, can be estimated by taking the product of the pseudoinverse of the matrix of input vectors and the desired output, both based on the data from the prior n observations.

We've been a little bit vague about what precisely n means in these equations. As it turns out, the nature of n depends on how we decide to approach modeling an environment. If we take a time-based approach and observe an environment for a duration of time, n, the value of n will be a time instant during the observation. The behavior of the environment would be computed as the mean of the results from all the time points in the duration. Alternatively, we can look at the behavior of multiple copies of the model all acting with the same input vector. In this case, the results of all the copies form an **ensemble**, or set of results, and we would use the mean of the ensemble to represent the behavior of the environment. Both approaches can yield different results. However, if the environment is **ergodic**; i.e., the statistics of the environment can be inferred from a single, sufficiently long random sample, we can use ensemble averages instead of time averages. The advantage of doing this is that it is generally easier to test multiple copies of the model for a shorter sample size than it is to wait for a single copy of the model to work over an extended period of time.

The value, $\mathbf{w}(n + 1)$, is the weight vector for the next observation (or time point) that we are trying to predict. We want to get as close to the optimal value for this as possible. In particular we want the following:

$$\mathbf{w}_o = \lim_{n \to \infty} \mathbf{w}(n+1)$$

where $\mathbf{w}_o$ is the **Wiener solution**, and n is the number of observations or time points. What the above equation states is that, *the more observations or time points we use, the closer we will get to the optimal value for* $\mathbf{w}$.

One drawback to using the Wiener filter is that the statistics required to compute the adjusted weight vector may not be available, for example if the environment is unknown. The Wiener filter relies heavily on knowledge of the environment in order to be effective. Additionally, while it is intuitive that more observations should produce a more accurate estimate of the weight vector, we may only have limited data to go on, and in that case the Wiener filter may not be able to produce a good result.

6.6 The Least-Mean-Square Algorithm

As we mentioned, one of the disadvantages of the Wiener filter is that it doesn't really work if the statistics of the environment are unknown. In order to have such statistics, it is necessary to observe the environment's behavior for a sufficient length of time, or be able to carry out a sufficient number of observations. When the statistics of the environment are unknown, we have to rely on estimating the weight vector based only on data from a single point in time, n. The **least-mean-square (LMS) algorithm** is designed to do precisely that. The LMS algorithm tries to minimize the following cost function at one instant in time:

$$E(\hat{\mathbf{w}}) = \frac{1}{2} e^2(n)$$ where $e(n)$ is the error signal measured at time n

The '^' symbol above the $\mathbf{w}$ indicates an instantaneous estimate for the weight vector, $\mathbf{w}$. The term is prnounced as "$\mathbf{w}$ *hat*". This formula for the LMS algorithm can also be written in terms of the steepest descent formula, as follows:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \eta \mathbf{x}(n) e(n)$$

This formula states that the estimate for the weight vector at a particular instant in time, $n + 1$, is given by the sum of the weight vector at the preceding time, n, and the product of the input vector and error signal at time n. This latter product is also modified by a learning rate parameter, η, which acts as a measure of the **memory** of the algorithm. This memory value is a scalar, not a vector or matrix, so it is only an artificial representation of memory, in contrast to the prior knowledge that is used by the Wiener filter. Smaller values for η represent *longer* memory, and results in more accurate estimates, but the tradeoff is slower convergence to the optimal value for the weight vector. Conversely, larger values for η represent *shorter* memory and less accurate estimates, but faster convergence. This is not meant to imply that one should always choose larger values for η, since although the convergence will be faster, the algorithm may not be fine-tuned enough to hit the optimal weight vector.

The behavior of the LMS algorithm is interesting in that if you were to plot the results you would see that the LMS algorithm does not follow a well defined trajectory like the steepest descent algorithm does. Instead, it randomly "walks" in proximity to the Wiener solution. Thus, the LMS algorithm is stochastic in its behavior, but this does not pose a serious problem in the algorithm's ability to estimate the optimal weight vector. Any inefficiencies that do result are more than offset by the lack of a requirement for statistical knowledge of the environment.

6.7 Pros and Cons of the LMS Algorithm

None of the algorithms we have discussed thus far, or that we will discuss in this course are perfect. Each one has its advantages and disadvantages, and the trick to deciding which algorithms to use will depend on a variety of factors. Below I summarize some of the pros and cons of the LMS algorithm.

Computational Complexity

All things considered, the LMS algorithm has remarkably low computational complexity compared to other algorithms that perform the same type of function. It is linear in the number of adjustable parameters; i.e., the weight values in the weight vector. Haykin mentions the algorithm can be coded using only 2-3 lines of code, but he doesn't elaborate on this at all. However, a O(n) algorithm that can be easily coded is certainly a point in favor of the LMS algorithm.

Robustness

In general, adaptive filter algorithms are sensitive to certain **disturbances**, such as the noise vector that represents the noise being filtered out of the environment, and the initial guess for the value of the weight vector. These disturbances are more pronounced the fewer the number of iterations of the algorithm. As the number of iterations approaches infinity, the effect of the disturbances approaches optimal minimization, and the desired output is maximized. With respect to the way robustness is defined here, the LMS algorithm is highly robust compared to other adaptive filters.

Slow Rate of Convergence

The primary disadvantage of the LMS algorithm is its slow rate of convergence to the optimal weight vector value. The algorithm requires a number of iterations that is approximately 10 times the dimensionality of the input data space just to reach a stable, steady-state condition. Thus, the higher the dimensionality of the input space, the slower the convergence.

In summation, the LMS algorithm has proven to be a relatively simple yet effective adaptive filter, and has been used for decades in a wide range of applications. Many variants of the algorithm have been developed since the algorithm's inception, with each variant attempting to improve upon some aspect of the original algorithm's performance, or to adapt the algorithm to a specific application. Four of the better known variants are:

- **LMS-Newton** (predictable convergence, but increased computational complexity)
- **NLMS** (Normalized LMS; better performance and tracking, at cost of increased risk of misadjustment (difference between Wiener performance and actual performance))
- **TD-LMS** (Transform Domain LMS; good tracking in non-stationary environments)
- **APA** (Affine Project Algorithm; better than NLMS when input signal is highly correlated; increased computational complexity)

References

1. Widrow, B. and Hoff, M.E. 1960. "Adaptive switching circuits", *IRE WESCON Conv. Rec.*, pp. 96-104.

2. McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133.