This Part 3 is a copy of Section 6 of my legacy lecture notes on Machine Learning (full set at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/lectureNotes/LN_ML_Fall11.pdf). I did not change the enumeration of equations, which all start with "6. ..." because this material was originally Section 6.

Version history:

V0.1 (posted April 7): direct copy from old LNs
V0.2 (posted April 8): weeded out dangling references, deleted some text that pointed to other parts of the original LNs
V0.3: completed broken sentence page 6, several sentences in Section 6.2 stated more clearly
6 The adaptive linear combiner

Overview. In this section we will consider a linear regression task (as opposed to the linear classification tasks from Section 4) on time series data (as opposed to the static samples $x$ from Section 4), training an online adaptive filter that incrementally adapts its weights as new data come in (as opposed to the "batch" offline least squares solutions from Section 4). The background is signal processing / electrical engineering. The assumption that the dynamic system we want to learn is indeed linear is often satisfied in signal processing – wired and wireless communication channels, which are a main type of system that require signal processing techniques, are indeed typically quite linear.


6.1 The adaptive linear combiner, a special Wiener filter

I start with a refresher on systems and signals terminology. A discrete-time, real-valued signal is a left-right infinite sequence $x = \{x(n)\} = (x(n))_{n \in \mathbb{Z} \subseteq \mathbb{R}}$. (We will only consider discrete-time, real-valued signals here.) Note that in this section, $x$ and $\{x(n)\}$ refer to a complete sequence of values; if we want to single out a particular signal value, we write $x(n)$. A system (or filter) $H$ is a mapping from signals to signals, written $y = H(x)$ or in graphical notation

![Signal Processing Diagram](image)

The signal $x$ is the input signal and $y$ is the output signal of $H$. A system is linear if for all complex constants $a$, $b$ and signals $x_1$, $x_2$ it holds that

$$H(a \, x_1 + b \, x_2) = H(a \, x_1) + H(b \, x_2).$$

A system is shift-invariant if

$$\forall \{x(n)\} \forall k \in \mathbb{Z} \quad H(\{x(n-k)\}) = \{H(\{x(n)\})(n-k)\}.$$

A shift-invariant, linear system is called an LSI system for short. We will be concerned with LSI systems exclusively.

- The unit impulse $\delta(n)$ is a signal that is defined to be 1 for $n = 0$ and zero elsewhere. Let $H(\delta) = h$ be the impulse response of a system $H$. For an LSI $H$, we get the system response on input $x$ by convolving $x$ with the impulse response:

$$\{y(n)\} = \left\{ \sum_{k=-\infty}^{\star} x(k) \, h(n-k) \right\} = \left\{ \sum_{k=-\infty}^{\star} h(k) \, x(n-k) \right\} = \{x(n)\} \otimes \{h(n)\}.$$

A system is causal if its current output does not depend on future inputs, or equivalently, if $h(n) = 0$ for $n < 0$. 

An LSI filter is a *finite impulse response filter* (FIR filter) if $h$ has a finite carrier, that is, $h$ is zero except at a finite number of points.

We will restrict ourselves to time-domain signal representations in this section; a frequency-domain treatment of adaptive filters is also possible but seems less common (Farhang-Boroujeny book, Section 7).

In a causal FIR filter, the output $y(n)$ is a linear function of a finite number of previous $M$ inputs $x(n), x(n - 1), ..., x(n - M + 1)$, that is,

$$y(n) = \sum_{j=1}^{M} w_j x(n - j + 1).$$

(6.4)

Engineers call this equation a *transversal filter*, and the engineer's (and Simulink's) way of graphically representing it is shown in Figure 6.1.

![Figure 6.1: A transversal filter (black) and an adaptive linear combiner (black plus blue). The $z^{-1}$ boxes are unit delay elements.](image)

We will consider the task where the filter output $y(n)$ should be made to follow a *desired* teacher signal $d(n)$ as well as possible in the mean square error sense, by adapting the weights $w_j$. Engineers sometimes call this setup an *adaptive linear combiner* (blue parts in Fig. 6.1), a special case of *Wiener filters* (Wiener filters in general have a version of Eq. (6.4) where on the lhs there are also weighted terms of the form $w_j y(n-k)$, that is, the current output depends on previous inputs and previous outputs. If such terms are included, the impulse response of a filter generally attains infinite length, and one has *infinite impulse response* (IIF) filters).

Another name for transversal filters is *tapped delay line*, and the filter weights are sometimes called *tap weights*.

Formally, we want to find optimal weights $w_{\text{opt}} = (w_{\text{opt}1}, ..., w_{\text{opt}M})^T$ such that
(6.5) \( w_{\text{opt}} = \arg \min \limits_w E[\varepsilon_w(n)^2] = \arg \min \limits_w E[(d(n) - y_w(n))^2] = \arg \min \limits_w E[(d(n) - w^T x(n))^2], \)

where \( x(n) = (x(n), x(n-1), ..., x(n-M+1))^T. \)

This is a linear regression task with training argument-value pairs \((x(n), d(n))\). We could use any algorithm for linear regression to obtain an estimate for \( w_{\text{opt}} \) from a finite training data set comprising filter inputs \( x(1), ..., x(N) \) and desired outputs \( d(M), d(M+1), ..., d(N) \).

However, here we want to derive an online, adaptive algorithm that updates the weights incrementally as new training data \((x(n), d(n))\) come in. Such an adaptive procedure maintains a set of weights \( w(n) \) at every time, and should yield the correct optimal weights in the limit of infinite time, that is, \( \lim_{n \to \infty} w(n) = w_{\text{opt}} \). This reflects the temporal nature of our training data, and the common situation in signal processing that a filter should be able to track time-varying systems online.

**Wiener-Hopf equation.** In order to prepare the grounds for an online learning algorithm, we derive a particular solution to the linear regression task, the Wiener-Hopf equation, from scratch. Let \( \xi(w) = E[(\varepsilon_w(n))^2] \) denote the mean square error, and rewrite it as follows:

\[
\begin{align*}
\xi(w) &= E[(d(n) - w^T x(n))(d(n) - w^T x(n))] \\
&= E[(d(n))^2] - 2 w^T E[x(n)d(n)] + w^T E[x(n)x^T(n)]w \\
&= E[(d(n))^2] - 2w^T p + w^T R w,
\end{align*}
\]

where we introduced the \( M \times 1 \) cross-correlation vector of the tap inputs with the desired signal

\[
(6.7) \quad p = E[x(n)d(n)] = (E[x(n)d(n)] ... E[x(n-M+1)d(n)])^T,
\]

and the \( M \times M \) correlation matrix

\[
(6.8) \quad R = E[x(n)x^T(n)].
\]

Eq. (6.6) is a quadratic function in \( w \). Because \( \xi(w) \) cannot be negative, \( \xi(w) \) must have the shape of a hyperparabeloid which is opened upwards. Figure 6.2 shows this function for the case of two-dimensional \( w \).
The function shown in Fig. 6.2 is called the performance surface of an adaptive linear combiner. It has a minimum at $\mathbf{w}_{\text{opt}}$, which is the unique weight value where the gradient of $\xi$ vanishes. This gradient can be computed (by expanding (6.6), for a complete derivation see Farhang-Boroujeny p. 53) as

$$
\nabla \xi = \begin{pmatrix} \frac{\partial \xi}{\partial w_1} & \ldots & \frac{\partial \xi}{\partial w_M} \end{pmatrix}^T = 2\mathbf{R}\mathbf{w} - 2\mathbf{p}.
$$

Putting this to zero gives us the Wiener-Hopf equation

$$
\mathbf{R} \mathbf{w}_{\text{opt}} = \mathbf{p},
$$

which yields the optimal weights by $\mathbf{w}_{\text{opt}} = \mathbf{R}^{-1}\mathbf{p}$.

**Principle of orthogonality.** In passing, we derive a fundamental property of optimally tuned transversal filters, the principle of orthogonality, which states that the residual error is uncorrelated to all tap inputs. While we will not use this principle in the sequel, it provides a deeper insight into such filters and is often exploited in the analysis and design of optimal filters.

Using $\xi = E[(\epsilon^2(n))]$ we have

$$
\frac{\partial \xi}{\partial w_i} = E \left[ 2\epsilon(n) \frac{\partial \epsilon(n)}{\partial w_i} \right] \quad \text{for} \ i = 1, \ldots, M.
$$
Since \( \varepsilon(n) = d(n) - y(n) \) and \( d(n) \) does not depend on \( w_i \), it holds that

\[
\frac{\partial \varepsilon(n)}{\partial w_i} = -\frac{\partial y(n)}{\partial w_i} = -x(n-i+1).
\]

Inserting this into (6.11) yields

\[
\frac{\partial \xi}{\partial w_i} = -2E[\varepsilon(n)x(n-i+1)].
\]

For optimal weights these gradients are zero, that is,

\[
0 = E[\varepsilon_{opt}(n)x(n-i+1)] \quad \text{for } i = 1, ..., M,
\]

which is the principle of orthogonality. Intuitively, it can be re-phrased like this: "As long as there is any correlation between a tap input and the current error, one can reduce the error further by subtracting away this correlation through a suitable tuning of the weights".

A geometric interpretation of the principle of orthogonality is maybe more enlightening than this rote derivation. A signal source for signals induces an inner product on such spaces, thereby a norm. A sequence of random variables \( X_\omega \), where \( X_\omega \) is an example of a stochastic process. Random variables \( X_n, Y_m, \) etc., can be linearly combined and thus be conceived as vectors in a suitable vector space \( V \) (these vectors are numerical functions from \( \Omega \) to \( \mathbb{R}^Z \)). Such vector spaces are typically infinite-dimensional. The correlation \( E[X Y] \) induces an inner product on such spaces, thereby a norm \( \|X\| = E[X^2]^{1/2} \) and thus a metric \( d(X,Y) = \|X - Y\| \), plus a notion of orthogonality: \( X \perp Y \) iff \( E[X Y] = 0 \), that is, two such random variables are orthogonal if they are uncorrelated. In short, we get all the conveniences of a (pre-)Hilbert space – that is, intuitively, you can work with random variables as with vectors of an Euclidean vector space. Now, let's reconsider our tapped delay line. The inputs \( x(n), x(n-1), ..., x(n-M+1) \) turn into random variables \( X_n, X_{n-1}, ..., X_{n-M+1} \), as does the teacher signal \( d(n) \) which becomes \( D_n \). All of these are vectors in \( V \). The vectors \( X_n, X_{n-1}, ..., X_{n-M+1} \) span an \( M \)-dimensional subspace \( S \) in \( V \). Typically, \( D_n \) is not contained in this subspace. The task of finding optimal weights, in this view, boils down to combine the filter input vectors into a filter output vector \( Y_n \) via \( Y_n = w_1X_n + ... + w_MX_{n-M+1} \), such that the error signal achieves minimal norm, that is, such that \( \|D_n - Y_n\| = E[(D_n - Y_n)^2]^{1/2} = \xi_n^{1/2} \) becomes minimal. Geometrically this amounts to finding the orthogonal projection of \( D_n \) on the subspace \( S \). The error signal \( \varepsilon = d - y \) becomes the vector \( E_n = D_n - Y_n \). We can simply re-use Figure 4.10 with different vector names to illustrate this. It becomes clear from Figure 6.3 that the optimal weights lead to an error vector that is orthogonal to all the signals \( X_n, X_{n-1}, ..., X_{n-M+1} \) – but this is just the principle of orthogonality.
6.2 Basic applications of adaptive linear combiners

In the previous subsection we considered the following general situation. A time series \( x = \{x(n)\} = (x(n))_{n \in \mathbb{Z}} \) of some filter inputs is given, together with a desired filter output \( d = \{d(n)\} = (d(n))_{n \in \mathbb{Z}} \). We started to address the task to train a filter that on the same input \( x \) produces an output \( y = \{y(n)\} = (y(n))_{n \in \mathbb{Z}} \) that matches \( d \) as closely as possible in the mean square error sense. We considered tapped delay line filters, but other, more complicated filter designs are of course also possible. Before we proceed with learning algorithms for this task, we will briefly present some standard application situations where this task arises. In the signal processing field, one often finds four basic applications: system identification, inverse system identification, adaptive noise cancelling, and beamforming (design of antenna arrays).

6.2.1 System identification

This is the most basic task: reconstruct from \( x \) and \( d \) a filter ("model system", "system model", "identification model") \( y \) that approximates \( d \). This kind of task is called system identification. A schematic block diagram for this kind of application looks as follows:

![System Identification Diagram](image-url)
Notes:

• The randomness that is inherent in most real-life systems is modeled by white noise $\nu$ that is added to a deterministic system output $g$. This is a highly simplifying assumption ("system + noise" model). Other models of randomness might for instance have systems whose parameters vary randomly – which leads to much more complicated maths.

• The graphical representation with the diagonal $\varepsilon$-arrow through the model system should be read as "adjust model parameters such that $E[|\varepsilon|^2]$ is minimized".

• If the unknown system is shift-invariant, the system identification means to find a model of the system. If however the unknown system is non-stationary, that is, its parameters vary (slowly) over time, the system identification task means that one wants to track the unknown system, that is, over time the model system should follow the unknown system as closely as possible.

Examples (taken from Farhang-Boroujeny).

1. Geological exploration. At one point $A$, the earth surface is excited by a strong acoustic signal $x$ (explosion or large vibrating mass). An earth microphone is placed at a distant point $B$, picking up a signal $d$. A model $M$ ("dummy earth") is learnt. After $M$ is obtained, one may analyse the impulse response $r$ of $M$. The peaks of $r$ give indications about reflecting layers in the earth crust between $A$ and $B$, which correspond to different delayed responses $p_i$ of the input signal $x$.
2. Adaptive open-loop control. In general terms, an open-loop (or direct or inverse or feedforward) controller is a device that generates an input signal \( u \) into a system (or plant) such that the system output \( y \) follows a reference (or target) trajectory \( r \) as closely as possible. In linear systems theory, the system is characterized by a transfer function \( H(\omega) = Y(\omega)/U(\omega) \) in the frequency domain (where \( U, Y \) are the frequency transforms of the input and output signals of the system, respectively). If the controller has a transfer function \( H^{-1}(\omega) = U(\omega)/Y(\omega) \), and the controller is serially connected to the plant, the two transfer functions cancel out and \( r = y \) is obtained. One way to obtain \( H^{-1} \) is to identify \( H \) online as an adaptive linear combiner and compute \( H^{-1} \) analytically, as shown in Fig. 6.6:

![Diagram of adaptive open-loop control](image)

**Figure 6.6:** Schema of online adaptive direct control.

### 6.2.2 Inverse system identification

This is the second most basic task: given an unknown system that on input \( d \) produces output \( x \), learn an inverse system that on input \( x \) produces output \( d \) [note the reversal of variable roles and names]. A typical setup is shown in Figure 6.7.
Introducing the delay $z^{-\Delta}$ is not always necessary but typically improves stability of the learnt system. Inverse system identification is also referred to as deconvolution because the original system $H$ transforms its input $d$ by convolving it with its impulse response $h$.

**Examples.**

**Equalization of a communication channel (from Farhang-Boroujeny).** A prime application of inverse system modelling is in telecommunications, where a binary signal $s$ is distorted when it is passed through a noisy channel $H$, and should be un-distorted ("equalized") by passing it through an equalizing filter with system transfer function $H^{-1}$. In order to train the equalizer, the correct signal $s$ must be known by the receiver, where the equalizer is trained. But of course, if $s$ would be already known, one would not need the communication in the first place... this hen-and-egg problem is often solved by using a predetermined training sequence $s = d$. From time to time (especially at the initialization of a transmission), the sender transmits $s = d$, which is already known by the receiver and enables it to estimate an inverse channel model. But also while useful communication is taking place, the receiver can continue to train its equalizer, as long as the receiver is successful in restoring the binary signal $s$: in that case, the correctly restored signal $\hat{s}$ can be used for continued training. The overall setup is sketched in Figure 6.8

**Figure 6.8:** Schema of adaptive online channel equalization. Delays are omitted.

**Feedback error learning for a composite direct / feedback controller.** Pure open-loop control cannot cope with external disturbances to the plant. Furthermore, the simple setup from Fig. 6.6 requires that for training the plant is driven by specially prepared training input,
a condition not desirable in true online applications where the controller has to adapt to the plant continuously while the entire system is operating. The following scheme (proposed by Michael Jordan in a nonlinear control context, using neural networks\(^1\)) trains an open-loop inverse controller in conjunction with the operation of a fixed feedback-controller. The architecture is shown in Fig. 6.9.

![Diagram of feedback error learning](image)

**Figure 6.9:** Schema of feedback error learning for a composite control system.

Some explanations on this ingenious architecture:

- The control input \(u(n)\) is the sum of the outputs \(u_{fb}(n)\) of the feedback controller and \(u_{ff}(n)\) of the feedforward controller.
- If the feedforward controller works perfectly, the feedback controller detects no discrepancy between the reference \(r\) and the plant output \(y\) and therefore produces a zero output \(u_{fb}(n)\) – that is, the feedforward controller sees zero error \(\varepsilon\) and does not change.
- If the feedforward controller does not work perfectly, the feedback controller's output \(u_{fb}(n)\) acts as an error signal for further adaptation of the feedforward controller. The feedforward controller tries to minimize this "error" – that is, it changes its way to generate output \(u_{ff}(n)\) such that the feedback controller's output is minimized, that is, such that \(r - y\) is minimized, that is, such that the control improves.
- When the plant characteristics change, or when external disturbances set on, the feedback controller sets on again – as does the further adaptation of the feedforward controller. Thus, situations that cannot be handled by a pure feedforward controller are cope with by the composite architecture, which is always operative.

### 6.2.3 Interference cancelling, "denoising" (from Farhang-Boroujeny)

Assume that there is a signal \(s + v_0\) that is an additive mixture of a useful signal \(s\) and a noise component \(v_0\). You want to cancel the interfering component \(v_0\) from this mixture. Assume further that you also have another signal source \(v_1\) that correlates strongly with \(v_0\) but weakly with \(s\). In this situation you may use a denoising scheme as shown in Figure 6.10.

---

Some explanations:

1. The "error" that the adaptive denoising filter tries to minimize is $s + \nu_0 - y$.
2. The only information that the filter has to achieve this is its input $\nu_1$. Because this input is (ideally) not correlated with $s$, but highly correlated with $\nu_0$, all that the filter can do is to subtract from $s + \nu_0$ whatever it finds correlates in $s + \nu_0$ with $\nu_1$. Ideally, this is $\nu_0$. Then, the residual "error" $\xi_{\text{min}}$ would be just $s$.
3. Note that the working principle behind this architecture is just an application of the principle of orthogonality.
4. This scheme is interesting (and not just a trivial subtraction of $\nu_1$ from $s + \nu_0$) because the correlation between $\nu_1$ and $\nu_0$ may be complex, involving superposition of delayed versions of $\nu_0$.
5. Applications include cleaning up EKG signals (the $\nu_1$ signal corresponds to electrodes that are planted on you at distant positions from the heart), distinguishing the child's heartbeat from the mother's in prenatal diagnosis, cancelling the 50Hz background noise found in many biological recordings, denoising of speech signals. Interference cancelling as explained here is a traditional technique. Today, one might want to employ the more advanced techniques of blind signal separation for similar purposes. But I would not be surprised if most EKG recording devices sold today still use this traditional approach.

6.2.4 Beamforming

I will only briefly mention the fourth traditional application area of adaptive filters. If one has an array of $M$ omnidirectional antennas, at which a mixture of $M$ radio signals $x_i$ arrives, all of the same frequency but coming from different directions, it is desirable in many telecommunication applications to pick out one of $M$ incoming signals from all the others, say $x_1$. This can be done by postprocessing the $M$ antenna signals by a filter that basically cancels the interfering signals $x_2, ..., x_M$. What is different here as compared to the other applications of adaptive filters considered so far is that here the data vector $x_1, ..., x_M$ used as input to the filter is not temporal but spatial. However, the mathematics remain the same. The name beamforming illustrates that by adaptation of its filter, the antenna array forms a "lobe" or "beam", that is an angular segment in the compass circle from which it effectively receives signals while suppressing signal input from directions outside the lobe.
6.3 Gradient descent for finding optimal weights in online adaptive tasks

The solution of the Wiener-Hopf equation provides an offline algorithm to compute optimal weights from a fixed training time series. In practice, however, one often desires an online algorithm that incrementally improves the weights. Specifically this is the case when the system that one wishes to model is varying over time. Then, adaptive algorithms are needed. The terms "online" and "adaptive" have slightly different meanings. "Adaptive" refers to the circumstance that the target system is time-varying and the model has to track the target system. This is typically done by using online algorithms but could, in principle, be done with an offline algorithm, too – over time, one would have to collect training sequences into a memory, and recompute the model from scratch on new training sequences. "Online" refers to algorithms that adapt their weights incrementally using each new data point as it comes in while the filter is being used. That is, at every time \( n \), a set of weights \( w(n) \) is computed, and typically \( w(n) \neq w(n+1) \). For adaptive system identification tasks, online methods are in most cases more natural, more elegant, computationally cheaper, and more precise.

In this subsection we provide an introduction to the simplest kind of online algorithms. They rest on the idea of gradient descent: at each time \( n \), go "downhill" on the performance surface a little bit in the steepest direction, just like a tired mountaineer. We will first treat this task from a theoretical perspective, assuming that the gradient is perfectly known (Subsection 6.3.1), and then describe a practical algorithm that estimates this gradient online. This algorithm, variously known as the LMS-algorithm ("least mean square", this name is common in signal processing), as stochastic gradient descent (common in machine learning) or as the [Widrow-Hoff] delta rule (in the biologically oriented neural network community). This multitude of names indicates that this algorithm has been re-discovered independently many times in different contexts, and it is certainly the simplest and likely the most widely used algorithm in adaptive signal processing. (I re-discovered it myself when I started to work my way into machine learning...) I lean on the treatment given in Farhang-Boroujeny, but any other book on neural networks, pattern recognition or adaptive signal processing will treat this subject, too.

6.3.1 Principles of gradient descent on quadratic performance surfaces

**Further properties of the performance surface; normalized coordinates.** Our goal in this section is to find online adaptive algorithms that incrementally adapt the weights \( w(n) \) such that the error decreases. Such algorithms (of which there are many) exploit the geometry of the performance surface. Therefore, next we investigate this geometric object more closely.

First we use (6.6) and the Wiener-Hopf equation (6.10) to write in various ways the expected residual error \( \xi_{\text{min}} \) that we are left with when we have found \( w_{\text{opt}} \):

\[
\xi_{\text{min}} = E[(d(n))^2] - 2w_{\text{opt}}^T p + w_{\text{opt}}^T R w_{\text{opt}}
\]

\[
= E[(d(n))^2] - w_{\text{eq}}^T p = E[(d(n))^2] - w_{\text{opt}}^T R w_{\text{opt}} = E[(d(n))^2] - p^T R^{-1} p.
\]

Next we present an alternative version of the error function \( \xi \). Observing that the paraboloid is centered on \( w_{\text{opt}} \), that is has "elevation" \( \xi_{\text{min}} \) over the weight space, and that the shape of the paraboloid itself is determined by \( w_{\text{opt}}^T R w \), we find that we can rewrite (6.6) as
\[
(6.16) \quad \xi = \xi_{\text{min}} + (w - w_{\text{opt}})^T R(w - w_{\text{opt}}) \\
= \xi_{\text{min}} + v^T R v,
\]

where we introduced shifted weight coordinates \( v = w - w_{\text{opt}} \). Differentiating (6.16) w.r.t. \( v \) yields

\[
(6.17) \quad \frac{\partial \xi}{\partial v} = \left( \frac{\partial \xi}{\partial v_1}, \ldots, \frac{\partial \xi}{\partial v_M} \right)^T = 2R v.
\]

From our discussion in Section 4.4 we obtain immediately the following insights. Since \( R \) is symmetric and positive semi-definite, we can write \( R = U^T D U = U D U^{-1} \), where \( U \) contains a set of orthonormal real eigenvectors in its columns and \( D \) is a diagonal matrix containing the corresponding eigenvalues, which are likewise real, and non-negative. Furthermore, the eigenvectors \( u_i \) of \( R \) lie on the central axes of the hyperellipsoid formed by the contour lines of the performance surface (see Fig. 6.2, red arrows). By left-multiplication of the shifted coordinates \( v = w - w_{\text{opt}} \) with \( U^T \) we get new normal coordinates \( \tilde{v} = U^T v \). The coordinate axes of the \( \tilde{v} \) system are in the direction of the eigenvectors of \( R \), and equation (6.17) becomes

\[
(6.18) \quad \frac{\partial \tilde{\xi}}{\partial \tilde{v}} = 2D \tilde{v} = 2(\lambda_1 \tilde{v}_1 \cdots \lambda_M \tilde{v}_M)^T,
\]

from which we get the second derivatives

\[
(6.19) \quad \frac{\partial^2 \xi}{\partial \tilde{v}^2} = 2(\lambda_1 \cdots \lambda_M)^T,
\]

that is, the eigenvalues of \( R \) are (up to a factor of 2) the curvatures of the performance surface in the direction of the central axes of the hyperparabeloid. We will shortly see that the most natural and simple adaptive learning algorithm, the LMS algorithm, depends in its efficiency critically on these curvatures.

The basic formula for taking a small step downhill along the gradient, thereby adapting \( w(n) \) to \( w(n+1) \), is

\[
(6.20) \quad w(n+1) = w(n) - \mu \nabla \xi(w(n)),
\]

where \( \mu \) is a stepsize parameter and \( \nabla \xi(w(n)) \) is the gradient of the performance surface at point \( w(n) \). In typical cases, \( \mu \) is set to values of 1/100 to 1/1000 – we will later learn to optimize this. We now analyze the convergence properties of the update rule (6.20). We will operate in the normal coordinates \( \tilde{v} = U^T v \) (remember \( v = w - w_{\text{opt}} \) and \( U^T \) was the matrix containing orthonormal eigenvectors of \( R \); further recall that \( R = U^T D U \) and \( D \) contains the eigenvalues \( \lambda_j \) of \( R \) on its diagonal). By some elementary transformations [use (6.18)] (6.20) turns into

\[
(6.21) \quad \tilde{v}(n+1) = (I - 2\mu D)\tilde{v}(n).
\]
Because $\mathbf{I} - 2\mu \mathbf{D}$ is diagonal, this can be split up into the components of $\tilde{v}$, yielding

$$
(6.22) \quad \tilde{v}_j(n + 1) = (1 - 2\mu \lambda_j)\tilde{v}_j(n) \quad (j = 1, \ldots, M).
$$

This is a geometric sequence. If started in $\tilde{v}_j(0)$, one obtains

$$
(6.23) \quad \tilde{v}_j(n) = (1 - 2\mu \lambda_j)^n \tilde{v}_j(0).
$$

The sequence $\mathbf{w}(n)$ converges to $\mathbf{w}_{opt}$ if $\tilde{v}_j(n)$ converges to zero for all $j$. (6.23) implies that this happens if and only if $|1 - 2\mu \lambda_j| < 1$ for all $j$. These inequalities can be re-written as $-1 < 1 - 2\mu \lambda_j < 1$ or equivalently,

$$
(6.24) \quad 0 < \mu < 1/\lambda_j.
$$

Specifically, we must make sure that $0 < \mu < 1/\lambda_{max}$, where $\lambda_{max}$ is the largest eigenvalue of $\mathbf{R}$. Depending on the size of $\mu$, the convergence behavior of (6.23) can be grouped in four classes which may be referred to as overdamped, underdamped, and two types of unstable. Figure 6.11 illustrates how $\tilde{v}_j(n)$ evolves in these four classes.

**Figure 6.11:** The development of $\tilde{v}_j(n)$ [plotted in the $y$-axis] vs. $n$ [x-axis]. The qualitative behaviour depends on the stepsize parameter $\mu$. a. Overdamped case: $0 < \mu < 1/(2\lambda_j)$. b. Underdamped case: $1/(2\lambda_j) < \mu < 1/\lambda_j$. c. Unstable with $\mu < 0$ and d. unstable with $1/\lambda_j < \mu$. All plots start with $\tilde{v}_j(0) = 1$.

We can find an explicit representation of $\mathbf{w}(n)$ if we observe that $\mathbf{w}(n) = \mathbf{w}_{opt} + \mathbf{v}(n) = \mathbf{w}_{opt} + \sum_{j=1}^{M} \mathbf{u}_j \tilde{v}_j(n)$, where the $\mathbf{u}_j$ are the orthonormal eigenvectors of $\mathbf{R}$. Inserting (6.23) gives us
(6.25) \( w(n) = w_{\text{opt}} + \sum_{j=1}^{M} \tilde{v}_j(0) u_j (1 - 2\mu \lambda_j)^n \).

This representation reveals that the convergence of \( w(n) \) toward \( w_{\text{opt}} \) is governed by an additive overlay of \( M \) exponential terms, each of which describes convergence in the direction of the eigenvectors \( u_j \) and is determined in its convergence speed by \( \lambda_j \) and the stepsize parameter \( \mu \). One speaks of the \( M \) modes of convergence with geometric ratio factors \( 1 - 2\mu \lambda_j \). If all eigenvalues are roughly equal, convergence rates are roughly identical in the \( M \) directions. If however two eigenvalues are very different, say \( \lambda_1 \ll \lambda_2 \), and \( \mu \) is small compared to the eigenvalues, then convergence in the direction of \( u_1 \) will be much slower than in the direction of \( u_2 \) (see Figure 6.12).

**Figure 6.12:** Two quite different modes of convergence (a.) vs. rather similar modes of convergence (b.). Plot shows contour lines of performance surface for two-dimensional weights \( w = (w_1, w_2) \). Violet dotted lines indicate some initial steps of weight evolution, starting from \( w(0) \).

Next we turn to the question how the error \( \xi \) evolves over time. Recall from (6.16) that \( \xi = \xi_{\text{min}} + v^T R v \), which can be re-written as \( \xi = \xi_{\text{min}} + \tilde{v}^T D \tilde{v} \). Thus the error in the \( n \)-th iteration is

(6.26) \( \xi(n) = \xi_{\text{min}} + \tilde{v}^T(n) D \tilde{v}(n) = \xi_{\text{min}} + \sum_{j=1}^{M} \lambda_j (1 - 2\mu \lambda_j)^n \tilde{v}_j(0)^2 \).

For suitable \( \mu \) (see (6.24)), \( \xi(n) \) converges to \( \xi_{\text{min}} \). Plotting \( \xi(n) \) yields a graph known as learning curve. (6.26) reveals that the learning curve is the sum of \( M \) decreasing exponentials (plus \( \xi_{\text{min}} \)). Figure 6.13 shows a three-mode learning curve for the case \( \xi_{\text{min}} = 0 \), where in **a.** \( \xi(n) \) is plotted on a linear scale and in **b.** in a logarithmic scale.
Each of the terms \((1 - 2\mu \lambda_j)^{2n}\) is characterized by a time constant \(\tau_j\) according to

\[
(1 - 2\mu \lambda_j)^{2n} = e^{\frac{-n}{\tau_j}}.
\]

If \(2\mu \lambda\) is close to zero, \(\exp(2\mu \lambda)\) is close to \(1 + 2\mu \lambda\) and thus \(\ln(1 - 2\mu \lambda) = -2\mu \lambda\). Using this approximation, solving (6.27) for \(\tau_j\) yields for the \(j\)-th mode a time constant of

\[
\tau_j = \frac{1}{4\mu \lambda_j}.
\]

That is, the convergence rate (i.e. the inverse of the time constant) of the \(j\)-th mode is proportional to \(\lambda_j\) for very small \(\mu\).

However, this analysis is meaningless for larger \(\mu\). If we want to maximize the speed of convergence, we should use significantly larger \(\mu\), as we will presently see. As can be seen from Fig. 6.13 b., the final rate of convergence is dominated by the slowest mode of convergence, which is characterized by the geometrical ratio factor

\[
\max\{\|1 - 2\mu \lambda_j\|, j = 1, ..., M\} = \max\{\|1 - 2\mu \lambda\max\|, \|1 - 2\mu \lambda\min\|\}.
\]

In order to maximize convergence speed, the learning rate \(\mu\) should be chosen such that (6.29) is minimized. Some elementary considerations reveal that this minimum is attained at \(\|1 - 2\mu \lambda\max\| = \|1 - 2\mu \lambda\min\|\), which is equivalent to

\[
\mu_{\text{opt}} = \frac{1}{\lambda\min + \lambda\max}.
\]

For this optimal learning rate, \(1 - 2\mu_{\text{opt}} \lambda\min\) is positive and \(1 - 2\mu_{\text{opt}} \lambda\max\) is negative, corresponding to the overdamped and underdamped cases shown in Figure 6.11. However, the two modes converge at the same speed (and all other modes are faster). Concretely, the optimal speed of convergence is given by the geometric ratio facto
where the last term is found by substituting (6.30). This has a value between 0 and 1. There are two extreme cases: if \( \lambda_{\text{max}} = \lambda_{\text{min}} \), then \( \beta = 0 \) and we have convergence in a single step. As the ratio \( \lambda_{\text{max}} / \lambda_{\text{min}} \) increases, \( \beta \) approaches 1 and the convergence slows down toward stillstand. The ratio \( \lambda_{\text{max}} / \lambda_{\text{min}} \) thus plays a fundamental role in limiting the convergence speed of steepest descent algorithms. It is called the \emph{eigenvalue spread}. 

The eigenvalue spread is closely related to the spectral properties of the input process \( x \). We can only sketch the connection here. Recall that for a stationary stochastic process \( \{x(n)\} \), \( \Phi_{xx}(k) = E[x(n)x(n-k)] \) is the \emph{autocorrelation function} and \( \Phi_{xx}(\omega) = \sum_{k=-\infty}^{\infty} \phi_{xx}(k)e^{-j\omega k} \) is its \emph{power spectral density} (or simply \emph{power spectrum} or just \emph{spectrum}). For each frequency \(-\pi \leq \omega < \pi\), \( \Phi_{xx}(\omega) \) gives the squared contribution of that frequency (the \emph{energy} of that frequency) to \( x \). It can be shown (details in Farhang-Boroujeny p. 97ff) that

\[
\begin{align*}
\lambda_{\text{min}} & \geq \min_{\omega} \Phi_{xx}(\omega), \\
\lambda_{\text{max}} & \leq \max_{\omega} \Phi_{xx}(\omega).
\end{align*}
\]

Thus, if \( x \) has a flat power spectrum (i.e., \( \min_{\omega} \Phi_{xx}(\omega) = \max_{\omega} \Phi_{xx}(\omega) \)), then \( \lambda_{\text{max}} / \lambda_{\text{min}} \approx 1 \) and we can expect fast convergence in steepest descent algorithms – and conversely, if \( x \) has a very uneven power distribution, steepest descent algorithms are likely to perform poorly. For this reason, it helps to speed up convergence if the input signal \( x \) is first passed through a \emph{whitening filter} that flattens its power spectrum, before it is used as input to an adaptive filter.

### 6.3.2 The LMS algorithm

The update formula (6.20) for steepest gradient descent, \( w(n+1) = w(n) - \mu \nabla \xi(w(n)) \), is not useful in practice because the gradient \( \nabla \xi(w(n)) \) is not known. Remember that \( \xi = E[\varepsilon^2] \) is the expected squared error of filter output \( y \) vs. teacher \( d \). Given filter weights \( w(n) \), we need to estimate the expected squared error \( \xi(w(n)) \) of the filter output generated by the filter with weights \( w(n) \) vs. the teacher \( d \). At first sight, what one needs to estimate an expected squared error is \emph{time} – namely, to observe the ongoing filtering with weights \( w(n) \) for some time and then approximate \( \xi(w(n)) = E[\varepsilon^2(w(n))] \) by averaging over the errors seen in this observation interval. But we don't have this time – because we want to update \( w(n) \) at every time step \( n \). One ruthless way out of this impasse is to just use the \emph{momentary} squared error as an approximation to its \emph{expected} value, that is, use

\[
\xi(w(n)) \approx \varepsilon^2(w(n)) = (d(n) - w^T(n)x(n))^2.
\]

Using this most brutal possible approximation, the update formula (6.20) for steepest gradient descent becomes

\[
w(n+1) = w(n) - \mu \nabla \varepsilon^2(w(n)),
\]
We can compute $\nabla \varepsilon^2(w(n))$ as follows:

$$
\nabla \varepsilon^2(w(n)) = 2\varepsilon(w(n)) \nabla \varepsilon(w(n)) = 
2\varepsilon(w(n)) \left[ \frac{\partial \varepsilon(w(n))}{\partial w_1} \ldots \frac{\partial \varepsilon(w(n))}{\partial w_M} \right]^T
$$

(6.35)  

$$
= -2\varepsilon(w(n)) \left[ \frac{\partial y(n)}{\partial w_1} \ldots \frac{\partial y(n)}{\partial w_M} \right]^T [\text{use } \varepsilon(n) = d(n) - y(n)]
= -2\varepsilon(w(n)) \left[ x(n) \ldots x(n-M+1) \right]^T
= -2\varepsilon(n) x(n)
$$

where in the last step we simplified the notation $\varepsilon(w(n))$ to $\varepsilon(n)$. Inserting this into (6.34) gives

(6.36)  

$$
\begin{align*}
\textbf{w}(n+1) &= \textbf{w}(n) + 2 \mu \varepsilon(n) \textbf{x}(n),
\end{align*}
$$

which is the weight update formula of the LMS algorithm. This formula can hardly be beaten in simplicity and computational efficiency! For completeness, here are all the computations needed to carry out one full step of online filtering & weight adaptation with the LMS algorithm:

(4) read in input and compute output: $y(n) = \textbf{w}^T(n) \textbf{x}(n)$,
(4) compute current error: $\varepsilon(n) = d(n) - y(n)$,
(4) compute weight update: $\textbf{w}(n+1) = \textbf{w}(n) + 2 \mu \varepsilon(n) \textbf{x}(n)$.

One fact about the LMS algorithm should always be kept in mind: being a stochastic version of steepest gradient descent, the LMS algorithm inherits the problems connected with the power spectrum of the input process $x$. If this power spectrum is very unevenly distributed, the LMS algorithm is likely not to work satisfactorily. (As an aside, in my working with neural networks, I tried out learning algorithms related to LMS. But the input signal to this learning algorithm had an eigenvalue spread of $10^{-14}$ to $10^{16}$, so the beautifully simple LMS algorithm was entirely useless.)
Because of its eminent usefulness (if the input signal has a reasonably flat power spectrum), the LMS algorithm has been analysed in minute detail. We conclude this section by reporting the most important insights without mathematical derivations. At the same time we introduce some of the standard vocabulary used in the field of adaptive signal processing.

We assume that $x$ and $d$ are stationary processes. The evolution $w(n)$ of weights is now also a stochastic process, because the LMS weight update depends on the stochastic vector $x(n)$. One interesting question is how fast the LMS algorithm converges in comparison with the ideal steepest gradient descent "algorithm" $\bar{v}(n + 1) = (I - 2\mu D)\bar{v}(n)$ from (4)(6.21). Because we now have a stochastic update, the vectors $\bar{v}(n)$ become random variables and one can only speak about their expected value $E[\bar{v}(n)]$ at time $n$. [Intuitively, this value would be obtained if many (infinitely many in the limit) training runs $\omega$ of the adaptive filter would be carried out and in each of these runs, the value of $\bar{v}(n)$ at time $n$ would be taken, and an average would be formed over all these $\bar{v}(n)$.] The following can be shown (using some additional assumptions, namely, that $\mu$ is small and that the signal $x$ has no substantial autocorrelation for time spans larger than $M$):

$$E[\bar{v}(n + 1)] = (I - 2\mu D) E[\bar{v}(n)].$$

Rather to our surprise, if the LMS algorithm is used, the weights converge – on average – as fast to the optimal weights as when the ideal algorithm (4)(6.21) is employed. Figure 6.14 depicts an overlay of the deterministic development of weights according to (4)(6.21) (grayish pink line) with one run of the stochastic gradient descent according to the LMS algorithm.

![Figure 6.14: Illustrating the similar performance on average of deterministic (pink) and stochastic gradient descent.](image)

The fact that on average the weights converge to the optimal weights (cf. (6.6)) by no means implies that $\xi(n)$ converges to $\xi_{\text{min}}$. To see why, assume that at some time $n$, the LMS algorithm actually would have found the correct optimal weights, that is, $w(n) = w_{\text{opt}}$. What would happen next? Well, due to the random weight adjustment, these optimal weights would become misadjusted again in the next time step! So the best one can hope for asymptotically is that the LMS algorithms lets the weights $w(n)$ jitter randomly in the vicinity of $w_{\text{opt}}$. But this means that the effective best error that can be achieved by the LMS algorithm in the asymptotic limit is not $\xi_{\text{min}}$ but $\xi_{\text{min}} + \xi_{\text{excess}}$, where $\xi_{\text{excess}}$ comes from the random scintillations of the weight update. It is intuitively clear that $\xi_{\text{excess}}$ depends on the stepsize $\mu$ – the larger $\mu$, the larger we expect $\xi_{\text{excess}}$ to become. The absolute size of the excess error...
ξ_{excess} is not so interesting as is the ratio \( M = \frac{\xi_{excess}}{\xi_{min}} \), the relative size the excess error w.r.t. the minimal error. The quantity \( M \) is called the *misadjustment* and describes what fraction of the *residual error* \( \xi_{min} + \xi_{excess} \) can be attributed to the random oscillations effected by the stochastic weight update [i.e., \( \xi_{excess} \)], and what fraction is inevitably due to inherent limitations of the filter itself [i.e., \( \xi_{min} \)]. Notice that \( \xi_{excess} \) can in principle be brought to zero by tuning down \( \mu \) to zero – however, that would be at odds with the objective of fast convergence.

Under some assumptions (notably, small \( M \)) and using some approximations (cf. Farhang-Boroujeny, Section 6.3), the misadjustment can be approximated by

\[
(6.7) \quad M = \mu \text{trace}(R),
\]

where the *trace* of a matrix is the sum of its diagonal elements. The misadjustment is thus proportional to the stepsize and can be steered by setting the latter, if \( \text{trace}(R) \) is known. Fortunately, \( \text{trace}(R) \) can be estimated online from the sequence \( x(n) \) simply and robustly [how? – easy exercise].

This is an important insight if one wishes to track a nonstationary system adaptively while maintaining a given misadjustment. In this situation, one commits oneself to a fixed level of misadjustment, maintains an online estimate of \( \text{trace}(R) \), and uses \( \mu = \frac{M}{\text{trace}(R)} \).

Another issue that one has always to be concerned about in online adaptive signal processing is stability. We have seen in the treatment of the ideal case (Section 6.3.1) that the stepsize \( \mu \) must not exceed \( \frac{1}{\lambda_{max}} \) in order to guarantee convergence. But this result does not directly carry over to our stochastic version of gradient descent, because it does not take into account the stochastic jitter of the gradient descent, which is intuitively likely to be harmful for convergence. Furthermore, the value of \( \lambda_{max} \) cannot be estimated robustly from few data points in a practical situation. Using again middle-league maths and several approximations, in the book of Farhang-Boroujeny the following upper bound for \( \mu \) is derived:

\[
(6.8) \quad \mu \leq \frac{1}{3 \text{trace}(R)}
\]

If this bound is respected, the LMS algorithm converges stably.

In practical applications, one often wishes to achieve an initial convergence that is as fast as possible: this can be done by using \( \mu \) close to the stability boundary from (6.8). After some time, when a reasonable degree of convergence has been attained, one wishes to optimize the mismatch; then one switches into a control mode where \( \mu \) is adapted dynamically according to (6.7).

The LMS algorithm is since 40 years the workhorse of adaptive signal processing and numerous refinements and variants have been developed. Here are some:

4) An even simpler stochastic gradient descent algorithm than LMS uses only the sign of the error in the update, i.e. uses \( w(n+1) = w(n) + 2 \mu \text{sign}(e(n)) x(n) \). If \( \mu \) is a power of 2, this algorithm does not need a multiplication (a shift does it then) and is suitable for very high throughput hardware implementations. There exist yet other "sign-simplified" versions of LMS [cf. Farhang-Boroujeny p. 169]
5) Online stepsize adaptation: at every update use a locally adapted stepsize $\mu(n) = 1/(x^T(n) x(n))$. This is called "Normalized LMS" or "NLMS". In practice this pure NLMS is apt to run into stability problems; a safer version is $\mu(n) = \mu_0 /\|x(n)^T x(n) + \psi\|$, where $\mu_0$ and $\psi$ are hand-tuned constants [Farhang-B. p. 172]. In my own experience, normalized LMS sometimes works wonders in comparison with standard LMS.

6) Include a whitening mechanism into the update equation: $w(n+1) = w(n) + 2\mu R^{-1} \varepsilon(n) x(n)$. This "Newton-LMS" algorithm has a single mode of convergence, but a problem is to obtain a good estimate of $R^{-1}$. [Farhang-B. p. 210]

7) Block implementations: for very long filters (say, $M > 10,000$) and high update rates, even LMS may become too slow. Various computationally efficient "block LMS" algorithms have been designed in which the input stream is partitioned into blocks, which are processed in the frequency domain and yield weight updates after every block only ["block LMS", cf. Farhang-B. p. 247ff].

To conclude this section, it should be said that besides LMS algorithms there is another major class of online adaptive algorithms for tapped delay line filters, namely, recursive least squares (RLS) filters. RLS algorithms are not steepest gradient-descent algorithms; in fact, the background metaphor of RLS is not to minimize $\xi$ but to minimize the error $\zeta(n) = \sum_{i=-1}^{\infty} (d(i) - y(i))^2$, so the performance surface we know from LMS plays no role for RLS.

The main advantages and disadvantages of LMS vs. RLS are:

- LMS has computational cost $O(M)$, where $M$ is filter length; RLS has $O(M^2)$. Also the space complexity of RLS is an issue for long filters because it is $O(M^2)$.
- LMS is numerically robust, RLS is plagued by numerical stability problems.
- RLS has a single mode of convergence and converges faster than LMS, much faster when the input signal is highly coloured.
- RLS is more complicated than LMS and thus more difficult to implement.
- In applications where fast tracking of highly nonstationary systems is required, LMS may have better tracking performance than RLS.

The RLS class of algorithms has been boosted by the development of fast RLS algorithms which reach a linear time complexity in the order of $O(20M)$ [Farhang-B. Section 13].