System Identification

In this section, we will see how we can use least-squares to **learn** the impulse response of a linear time-invariant system by observing its input and output. We have seen already how we might set this up as a linear inverse problem (see the example at the beginning of Chapter II of these notes). Now, using the work we just did on recursive least-squares, we can describe how we could solve this problem in a streaming manner.

Specifically, suppose we observe the convolution

$$y[n] = \sum_{k=0}^{N-1} h_*[k]u[n-k] + \text{ noise},$$

where

- $u[n]$ is the input signal (observed),
- $y[n]$ is the output signal (observed), and
- $h_*[n]$ is the impulse response of (finite) length N (unknown).

Our goal is to estimate $\boldsymbol{h}_* \in \mathbb{R}^N$ after observing $u[n]$ and $y[n]$ over some amount of time.

Note that we can equivalently write our observations as

$$y[n] = \boldsymbol{A}_n \boldsymbol{h}_* \text{ noise},$$

where $\boldsymbol{A}_n$ is the $1 \times N$ matrix

$$\boldsymbol{A}_n = \begin{bmatrix} u[n] & u[n-1] & \dots & u[n-N+1] \end{bmatrix} = \boldsymbol{u}_n^{\mathrm{T}}.$$

With all of the measurement vectors up to time M stacked up:

$$\underline{\boldsymbol{A}}_M = \begin{bmatrix} \boldsymbol{A}_0 \\ \boldsymbol{A}_1 \\ \vdots \\ \boldsymbol{A}_M \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_0^{\mathrm{T}} \\ \boldsymbol{u}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{u}_M^{\mathrm{T}} \end{bmatrix}, \quad \underline{\boldsymbol{y}}_M = \begin{bmatrix} y[0] \\ y[1] \\ \vdots \\ y[M] \end{bmatrix},$$

we could then form the least-squares estimate

$$\widehat{\boldsymbol{h}} = (\underline{\boldsymbol{A}}_M^{\mathrm{T}} \underline{\boldsymbol{A}}_M)^{-1} \underline{\boldsymbol{A}}_M^{\mathrm{T}} \underline{\boldsymbol{y}}_M.$$

Alternatively, we could apply the recursive least-squares algorithm from page 55 of these notes, giving us the following iteration (we have moved things around to make things as efficient as possible for this special case):

$$\begin{aligned} \boldsymbol{v}_n &= \boldsymbol{P}_{n-1} \boldsymbol{u}_n \\ \boldsymbol{k}_n &= \frac{1}{1 + \boldsymbol{u}_n^{\mathrm{T}} \boldsymbol{v}_n} \boldsymbol{v}_n \\ \boldsymbol{h}_n &= \boldsymbol{h}_{n-1} + (y[n] - \boldsymbol{u}_n^{\mathrm{T}} \boldsymbol{h}_{n-1}) \boldsymbol{k}_n \\ \boldsymbol{P}_n &= \boldsymbol{P}_{n-1} - \boldsymbol{k}_n \boldsymbol{v}_n^{\mathrm{T}} \boldsymbol{P}_{n-1}. \end{aligned}$$

Note that this algorithm is much more efficient than re-solving the entire least-squares problem from scratch with each new measurement, but it still requires a matrix-vector multiplication at each iteration. It might be hard to imagine anything that would require even less computation, but there are some surprisingly effective alternatives that are even cheaper! We will talk about one particularly important example.

Least Mean Squares (LMS) Filter

The **Least Mean Squares (LMS) filter** is an important algorithm that is even cheaper (computationally) than recursive least-squares. This algorithm, introduced in 1960, is also the first example of a **stochastic gradient** algorithm, which is currently a hot topic in large-scale machine learning.

One way to think of the LMS filter is as a (very rough) approximation to using steepest descent to solve our least-squares optimization problem. To be more precise, recall that the optimization problem

$$\underset{\boldsymbol{h}\in\mathbb{R}^N}{\text{minimize}} \ \|\underline{\boldsymbol{A}}_M\boldsymbol{h} - \underline{\boldsymbol{y}}_M\|_2^2.$$

can be solved via an iterative update of the form

$$\boldsymbol{h}_{k+1} = \boldsymbol{h}_k + \alpha_k \boldsymbol{r}_k,$$

where

$$\boldsymbol{r}_k = \underline{\boldsymbol{A}}_M^{\mathrm{T}}\underline{\boldsymbol{y}}_M - \underline{\boldsymbol{A}}_M^{\mathrm{T}}\underline{\boldsymbol{A}}_M\boldsymbol{h}_k.$$

Note that we can also write

$$\boldsymbol{r}_k = \sum_{n=0}^{M}(y[n] - \boldsymbol{u}_n^{\mathrm{T}}\boldsymbol{h}_k)\boldsymbol{u}_n \tag{1}$$

Now, note that the most costly aspects of this algorithm are:

1. Computing the sum over all M terms in the gradient in (1), and
2. Running this algorithm to convergence (requiring many iterations) each time a new measurement $y[n]$ arrives.

The LMS filter does away with both of these restrictions by taking the following (seemingly crazy!) strategy: when a new measurement $y[n]$ arrives, take only a single gradient step, and ignore every term in (1) except the one corresponding to the most recent measurement! To be concrete, given $y[n]$ and an existing estimate $\boldsymbol{h}_n$, we produce an update via the iteration:

$$\boldsymbol{h}_{n+1} = \boldsymbol{h}_n + \alpha \left(y[n] - \boldsymbol{u}_n^T \boldsymbol{h}_n \right) \boldsymbol{u}_n.$$

Note that $\boldsymbol{u}_n^T \boldsymbol{h}_n$ is the output of your candidate filter whose taps are the current weights $\boldsymbol{h}_n$. We have also fixed the stepsize.

LMS Filter

Initialize: $\boldsymbol{h}_0 = \boldsymbol{0}$

for $n = 1, 2, 3, \ldots$ **do**

(observe input $u[n]$ and output $y[n]$)

$\boldsymbol{u}_n = \begin{bmatrix} u[n] & u[n-1] & \cdots & u[n-N+1] \end{bmatrix}^{\mathrm{T}}$

$\boldsymbol{h}_n = \boldsymbol{h}_{n-1} + \mu \left(y[n] - \boldsymbol{u}_n^T \boldsymbol{h}_{n-1} \right) \boldsymbol{u}_n$

end for

It can be shown that the LMS algorithm converges when the stepsize is sufficiently small (depending on the statistics of the signal $u[n]$).There are also many results that give the speed of convergence under various assumptions on the true $\boldsymbol{h}_*$ and $u[n]$. In general, these results say that to get within ϵ relative error, we need either $\sim 1/\epsilon$ or $\sim 1/\epsilon^2$ iterations (depending on the assumptions). Notice that this is dramatically worse than the $\sim \log(1/\epsilon)$ required for steepest descent.[1] But also notice that the iterations are much cheaper. Sim-

[1] Suppose $\epsilon = 10^{-3}$. What are $\log(1/\epsilon)$, $1/\epsilon$, $1/\epsilon^2$?

ilarly, the RLS algorithm tends to converge much more quickly than LMS, but each iteration is much more computationally expensive.

There is now a rich literature on the generalization of LMS where you implement steepest descent by, at each iteration, randomly selecting only a few of the terms in the sum that computes the gradient. This is now known as stochastic gradient descent (stochastic because the terms are typically chosen at random) and is a big part of the success of modern machine learning algorithms. (Computing the gradient in many algorithms involves taking a sum over every item in your dataset – if you start working with datasets of millions of images, speeding this up becomes critical!)

Adaptive Filtering

In many practical settings, the impulse response may gradually (or not so gradually!) drift over time. Both of the methods we have described above can handle this setting as well.

First, notice that the core update in the LMS filter depends only on the current output value and the last N input samples. This means that by approximating the gradient with only the last term, LMS naturally adapts to changing filter coefficients. The convergence results mentioned above can be very easily translated into characterizations of how closely we can track dynamic $\boldsymbol{h}_*$.

To make the RLS filter more agile to dynamic $\boldsymbol{h}_*$, we must find a similar way to slowly forget old inputs. This can be achieved by considering a regularized weighted least-squares problem. At time n,

we can choose $\boldsymbol{h}_n$ to solves

$$\underset{\boldsymbol{h}}{\text{minimize}} \ \sum_{m=0}^{n} \lambda^m \left| y[m] - \boldsymbol{u}_m^{\mathrm{T}} \boldsymbol{h} \right|^2 + \delta \lambda^n \|\boldsymbol{h}\|_2^2.$$

This is a Tikhonov-regularized least-squares problem, where the regularization parameter is getting smaller as n increases. We can again apply the matrix inversion lemma to obtain the following algorithm:

RLS Adaptive Filter

Initialize: $\boldsymbol{h}_0 = \mathbf{0}, \ \ \boldsymbol{P}_0 = \delta^{-1}\mathbf{I}$.

for $n = 1, 2, 3, \ldots$ **do**

(observe input $u[n]$ and output $y[n]$)

$\boldsymbol{u}_n = \begin{bmatrix} u[n] & u[n-1] & \cdots & u[n-N+1] \end{bmatrix}^{\mathrm{T}}$

$\boldsymbol{v}_n = \boldsymbol{P}_{n-1} \boldsymbol{u}_n$

$\boldsymbol{k}_n = \frac{1}{\lambda + \boldsymbol{u}_n^{\mathrm{T}} \boldsymbol{v}_n} \boldsymbol{v}_n$

$\boldsymbol{h}_n = \boldsymbol{h}_{n-1} + (y[n] - \boldsymbol{u}_n^{\mathrm{T}} \boldsymbol{h}_{n-1}) \boldsymbol{k}_n$

$\boldsymbol{P}_n = \lambda^{-1} \boldsymbol{P}_{n-1} - \lambda^{-1} \boldsymbol{k}_n \boldsymbol{v}_n^{\mathrm{T}} \boldsymbol{P}_{n-1}$

end for