ADAPTIVE EQUALIZATION NOTES

C. Rose

In your book, the concept of the zero forcing equalizer was explored. The overall model assumes an impulse put into a channel and the channel output $x(t)$ sampled at times nT to produce the discrete time sequence $x_n = x(nT)$. The zero-forcing equalizer is a FIR filter with tap weights w_k applied to the input $x(nT)$. Then, you looked to find tap weights which make the overall system impulse response (in discrete time) be a single lollipop at the origin and zero everywhere else (in a window to either side of size $\pm N$). That is, you want to make the system output $y_n = x_n * w_n = \delta(n)$ (for $n = -N, -N+1 \ldots 0, 1, 2, \ldots. N$). That $*$ means discrete convolution.

Assuming that the tails of the channel impulse response drop rapidly as compared to NT, this means that impulses put into the channel system at times kT have almost no effect on each other because the overall response to an impulse at kT is a value of $y_n = \delta(n-k)$ and the channel response outside a double-sided window of size N is assumed essentially zero. This roughly corresponds to the zero ISI condition where a pulse at 0 has no effect on pulses at times nT.

We found that the optimal tap weights could be found by inverting a matrix composed of channel impulse response samples between $\pm N$. Unfortunately, this is time consuming for larger N and it requires someone to tell us the channel characteristics (the channel impulse response) and when it changes significantly — bummer! All this is completely in the book, by the way.

We then moved to adaptive equalization. The problem here is slightly different. Rather than trying to force zero ISI by inverting matrices we formulated it as an error minimization problem. We have the same input sequence x_n to our equalizer with tap weights w_k, but this time we don't assume anything about the channel and simply try to find w_k which minimize an error criterion.

Well, in order to calculate error, we need to be pretty sure what the right answer is. So what we assume FIRST is that the transmitter is sending a known sequence of digits a_n over the channel. In the best of all possible worlds (read **Candide**) the output of the channel would be exactly a_n. However, evil is afoot and what we get is x_n. So what we want our equalizer to do is take the sequence x_n and produce (via discrete convolution) a sequence y_n. And to finish we want the y_n to be as close to the known a_n as possible!

This closeness is specified by an error function, our old friend mean square error $e_n^2 = (y_n - a_n)^2$ and we want this to be as close to zero as possible over all n.

So we form

$$\mathcal{E} = E[e_n^2] = E\left[(\sum_{k=-N}^{N} w_k x_{n-k} - a_n)^2 \right]$$

Now, $\mathcal{E}$ depends upon how well our equalizer weights are chosen. So what we want to do is figure out which weights minimize $\mathcal{E}$. We can therefore think of $\mathcal{E}$ as a function of the weight vector $\mathbf{w}$, or $\mathcal{E}(\mathbf{w})$.

Well, when we see "minimization" it's almost a reflex to take derivatives, set them to zero and declare extremum. The derivatives in question are

$$\frac{\partial \mathcal{E}}{\partial w_k} = -2E[e_n x_{n-k}] = -2R_{ex}(k)$$

where $R()$ is the cross correlation function between e and x. In a single dimension we know that the second derivative greater than zero everywhere implies convexity so that a unique minimum exists. Alas, we also know that for multivariate functions, the picture is not so rosey! More on this at the end!

The development in the text simply assumes that the minimum can be found by taking first derivatives and setting them to zero. It also develops a method of iteratively finding the minimizing weights by essentially "sliding down the error hill". This means that you modify your weights in a direction opposite to the gradient of the error surface. For example, say your initial guess is $\mathbf{w}(0)$. You modify your guess by forming for each k

$$w_k(1) = w_k(0) + 2\mu R_{ex}(k)$$

where μ is some suitably small "step size". You don't want the step size too small, otherwise you never get where you're going. Likewise with step size too large, you overshoot where you're going and may ping pong around but never reach the minimum. By the way, don't worry about the factor of 2 difference between the above expression and your book. That factor is easily subsumed in the μ.

The final practical whistle is to recognize that you can't really know the cross correlation function, so you fake it by making a guess as $\hat{R}_{ex}(k) = e_n x_{n-k}$, both factors which are available to you. You then modify your algorithm to

$$\hat{w}_k(1) = \hat{w}_k(0) + 2\mu e_n x_{n-k}$$

and at some point you stop when the error is small enough (stopping rule).

There's a neato keano interpretation for when you stop in the ideal case. That is, you stop when $R_{ex}(k) = 0$. Assuming that $E[x_n] = 0$ and $E[e_n] = 0$ that means

that you stop when the error and your observable x_n are UNCORRELATED. If we use uncorrelated as a surrogate for independence, that means we stop when the input sequence to our equalizer has no more to say about the error we generate! That is, the incoming observations add no new information about how to reduce the error and we've done as well as we can. This *principle of orthogonality* (orthogonal usually means independent when used in the random variable sense) is very useful and pops up all over the place in communications and estimation.

Well, that's all well and good, but remember that in order to calculate e_n we have to assume we know a_n. But if we always know a_n why are we bothering to send it? The answer is that we don't always send a known sequence. We only send that sequence while "training" the equalizer. Thus, the known sequence is called a "training sequence" for the equalizer and we make this training sequence long enough so that we're pretty sure we'll get close to the optimal weights $\mathbf{w}^*$ which minimize $\mathcal{E}$.

Once we're done training, however, we begin to send data. And here's the neat part! If we did a good job equalizing and the channel characteristics don't change rapidly, then our equalizer outputs y_n are pretty close to the unknown a_n. Let's assume that the a_n can take on values ± 1. If that's the case, then as the channel changes the equalizer outputs will begin to drift from ± 1. This will not immediatly affect our ability to correctly determine the a_n. For example, if the equalizer puts out $y_n = 0.8$ then we're probably safe guessing that $a_n = 1$.

HOWEVER, we have a nonzero e_n and we can use these small errors to correct the equalizer weights. The equalizer is then in "adaptive mode" as opposed to "training mode" and tracks slow channel changes by monitoring the error and adjusting the weights just as it did for training mode. The only difference is that in training mode the correct a_n were known exactly whereas in adaptive mode, the guesses at a_n are assumed accurate if we did a good job of training initially. Of course, if the channel changes too rapidly, then we suck at guesstimating the a_n and the weight updates will suck too. Then, the error begins to grow and the equalizer stops translating data and requests a drop back to training mode. If the channel has gotten so bad that it can't be equalized, then the line is dropped and you curse it soundly under your breath since this happened near the end of a large web page transfer.

So that's basically it. However, WE ARE ANALYSTS! Thus, there's LOADS of things we'd need to prove before we could even begin to believe that this REALLY works. We're not going to prove that the stochastic update algorithm (the weight update equation with all the hats in it) converges. The only thing we're going to prove is that setting the error derivatives equal to zero does guarantee we

find the optimal weights. To do this we're going to prove that $\mathcal{E}$ is convex in the weights w_k.

We have to fall back on the convexity definition from class. A function is convex if for $0 \leq \lambda \leq 1$ and any $\mathbf{x}_1$ and $\mathbf{x}_2$ we have

$$f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2)$$

This means that the surface always lies below the line drawn between any two points. It's pretty easy to show that in a single dimension, the old second derivative non-negative is exactly equivalent to this statement, but we'll forego that pleasure here.

Now, to the problem at hand. Our function is $E[(]\omega)$. To make life simpler we'll write $\sum_k w_k x_{n-k}$ as $\omega^T \mathbf{x}$. This leads to

$$E[(]\omega) = E[(a_n - \omega^T \mathbf{x})^2] = E[a_n^2 - 2a_n \omega^T \mathbf{x} + (\omega^T \mathbf{x})^2]$$

We then look at

$$E[(a_n - (\lambda\omega_1 + (1-\lambda)\omega_2)^T \mathbf{x})^2]$$

which is equivalent to

$$E[a_n^2 - 2a_n(\lambda\omega_1 + (1-\lambda)\omega_2)^T \mathbf{x} + |(\lambda\omega_1 + (1-\lambda)\omega_2)^T x|^2]$$

and try to see if it's less than

$$\lambda E[(a_n - \omega_1^T \mathbf{x})^2] + (1-\lambda) E[(a_n - \omega_2^T \mathbf{x})^2]$$

Well after playing around a bit (removing like terms from each side such as $E[a_n^2]$ on the left and right) and regrouping (like we did in class) we find that

$$E[(a_n - (\lambda\omega_1 + (1-\lambda)\omega_2)^T \mathbf{x})^2] \leq \lambda E[(a_n - \omega_1^T \mathbf{x})^2] + (1-\lambda) E[(a_n - \omega_2^T \mathbf{x})^2]$$

and the function $E[\omega]$ is indeed convex in ω.

TADA!!!!!!!

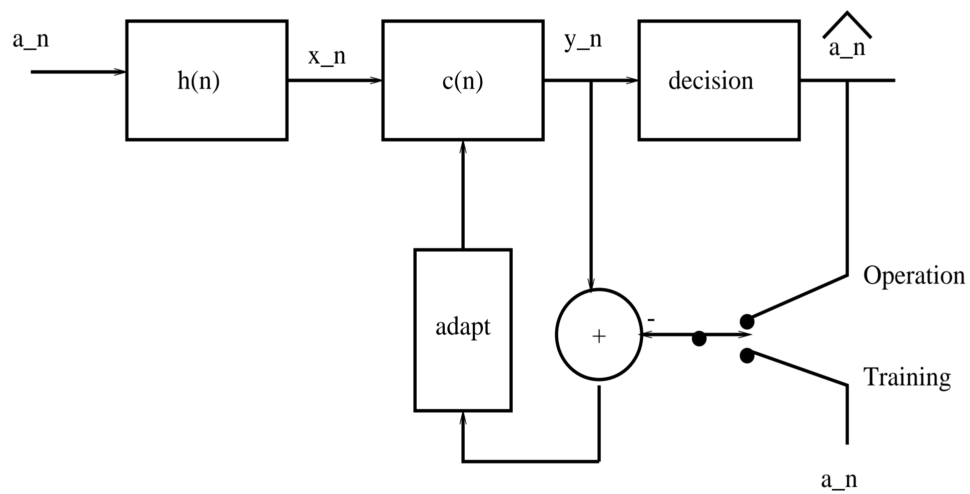

Figure 1: Here's a representative figure of the whole shebang