Belief Net

Intuitively, one problem with the simple version of relaxation labeling that I've sketched is seen if we have two lines, A and B, that support each other. At one iteration, A supports B. At the next, B supports A. When this happens, A is getting support for its being figure that really originates in A itself. It would be better if we could tell the difference between B seeming to be figure because of A's support, or support from another source. Belief nets formalize probabilistic support. Belief propagation is an algorithm that performs inference according to this probabilistic support.

******** This part omitted in '06

First, a belief net is a graphical representation of a set of random variables that makes explicit some properties about causality and conditional independence. As an example, suppose we have a house alarm. It can be set off by a burglar or an earthquake. If it goes off, a neighbor, Sally or John, might call you. The arrows show casuality. Conditional independence is also indicated; given that the alarm goes off, Sally and John calling are independent events. They're not independent if we don't know whether the alarm went off. The parent nodes have a different structure; the earthquake and burglary are independent if we don't know whether the alarm went off, but are not independent if we do know. That's because if the alarm goes off, knowing there was an earthquake effects my assessment of whether there was a burglary. To make this precise, we have to know the conditional probability of every variable, based on its causes.

We're going to talk about one simple algorithm for performing inference in these nets, that of belief propagation. Inference means that we are given some knowledge about the value of some variables, and want to determine the value of others. For example, in vision, we might know that some positions and orientations in the image have a contour going through them, and want to know whether others do. Though this algorithm is simple, it's very elegant and useful.

We now look at how one would really perform correct probabilistic inference in an image. We first look at the 1D case, which is substantially easier than the 2D case. Suppose we have an image that we think is piecewise constant, corrupt by noise. Then the noisefree intensity at each pixel depends on the observed intensity at that pixel, and also on the rest of the image. We make a Markov assumption that the true intensity of a pixel only directly depends on its observed intensity, and the true intensity of its neighbors, and is conditionally independent of everything else. This is what we would get if we generate an image by deciding independently at each pixel, whether to copy the previous pixel or jump to a new value, and then corrupting this with independent noise.

We describe this with a chain of variables that represent the true intensities

X1 -> X2 -> … Xn

And corresponding variables, yk, that indicate the observations. Collectively, we call all these e, for evidence. We want to compute the distribution of the true intensity at each pixel, conditioned on the observations. We do this by breaking the evidence into separate parts, which can all be handled independently.

Let's start with the simplest case, which will solve half the problem for us.

P(xn | e) = P(e/xn) P(xn) / P(e).

P(e) is the same for all values of xn; it's just a normalizing constant. So we can ignore it, and then normalize the values we get so they sum to one. P(xn) is the prior on xn.

P(e|xn) = P(e^- | xn, yn)*P(yn|xn)

e^- is defined to be all evidence to the left of xn. P(yn | xn) comes from our noise model, we assume we know all conditional probabilities.

P(e^- | xn, yn) = P(e^- | xn) \def \lambda(xn).

We compute this by taking advantage of conditional independence. If we knew xn-1, xn would then be independent of e^-. We don't know this, but we can sum over all possibilities:

\lambda(xn) = \sum_xn-1 P(e^- | xn, xn-1) * P(xn-1 | xn)

P(e^- | xn, xn-1) = P(e^- | xn-1) = \lambda (xn-1) * P(yn-1 | xn-1)

If we write P(xn-1 | xn) as a matrix, M_{xn-1 | xn} we get

\lambda(xn) = M_{xn-1 | xn} \lambda (xn-1) * componentwise multiplication with P(yn-1 | xn-1).

So we can compute probabilities recursively, starting at the left and working our way right.

If we want to handle a pixel that is not on the end, we can do this in much the same way, combining evidence from the left and from the right.

P(xk | e) = P(e^- | xk, yk, e^+) P(yk | xk, e^+) P(xk | e^+) * (P(e^+)/P(e))

The first term is just \lambda(xk). The second is the data term, P(yk|xk). The last is the normalizing constant. So we just need to figure out how to compute:

P(xk | e^+) \def \pi(xk)

This is almost the same deal as \lambda(xk). We get:

\pi(xk) = \sum_{xk+1} P(xk | xk+1, e^+) P(xk+1 | e^+) P(yk+1 | xk+1)

Again, the main computation is M_{xk | xk+1)*\pi(xk+1).

So our overall strategy is to pass information to the left and right, adding evidence as we go, and finally combine these by multiplying evidence from the left and right.