Boltzmann Machines

Sam Roweis

1 Stochastic Networks

Up till this point we have been studying associative networks that operate in a deterministic way. That is, given particular interconnections between units, and a particular set of initial conditions, the networks would always exhibit the same dynamical behaviour (going downhill in energy), and hence always end up in the same stable state. This feature of their operation was due to the fact that the rules of operation of each neuron had no probilistic elemets to their specification[1]. This feature was useful when we were interested in pattern completion and other kinds of explicit computaion for which we had some problem to be solved. The problem was encoded as the initial state of our network, our stored knowledge and constraints were encoded by the connections, and the solution was the final stable state into which the network settled.

Now we are going to consider another paradigm. Instead of using networks to find particular outputs for given inputs (e.g. associative memory recall or optimization problems), we want to use them to model the *statistical behaviour* of some part of our world. What this means is that we would like to show a network some distribution of patterns that comes from the real world and get it to build an internal model that is capable of generating that same distribution of patterns on its own. Such a model could be used to produce believable patterns if we need some more of them, or perhaps by examining the model we may gain some insight into the structure of the process which generated the original distribution. The more closely the probability distribution over patterns that the network generates matches the distribution in the real world, the happier we will be with our model.

Before we can go any further with this, I should introduce the idea of a network "generating" a distribution of patterns. The key new element is that now our rules of network operation must include some probabilistic steps. So now if a network is in a given state when we perform an update on one neuron, the next state of the neuron will not be given deterministically. Such a network can never settle into a stable state because units will always be changing state even if their inputs do not change. What good is that you ask ? Well,if we simply let such a network run freely for a long time and record the states it passes through, we can construct a probability distribution over those states. If we take those state vectors (or some subvectors of them) as our patterns, then we have solved the problem of how to make a network "generate" a probability distribution over a set of patterns.

To begin, consider a simple modification of the Hopfield net to use *stochastic units*. What did the original updating rule for the Hopfield net say ? It simply told each unit to switch into whichever of its states made the total energy of the system lower[2]. Luckily, if the connections between units were all symmetric, then each unit could make this decision *locally* by simply computing the energy difference ΔE between it being inactive and active. For the i^{th} unit this was simply:

$$\Delta E_i = E_i(-1) - E_i(+1) = \sum_j w_{ij} S_j$$

If ΔE was negative then it was better (lower system energy) to be inactive, otherwise it was better to be active. This gave our updating rule directly as in note 1.

We will now modify this updating rule to make it stochastic. Let each unit now set its new state to be *active* with probability:

$$p_i(+1) = \frac{1}{1 + e^{\Delta E_i / T}}$$

where T is a parameter that describes the "temperature" of the network. This ties into statistical physics

[1]For example, the rule for operating our Hopfield nets in the high gain limit was simply: (1)Pick a unit. (2)Compute its net input as the sum of connection weights to other *active* units. If this net input is positive, make the unit active (state=+1), if this net input is negative or zero make the unit inactive (state=-1).

[2]Remember our Lyapunov function $E = -\sum_{j,i<j} w_{ij} S_i S_j$ which was guarenteed to be *reduced* by each unit update ? Well this explains why: if unit k were to switch sign, it can compute the effect that would have on the global energy function by a purely local computation.

– the units will *usually* go into the state which reduces the system energy, but they will *sometimes* go into the "wrong state", just as a physical system sometimes (but not often) visits higher energy states. At zero temperature, this update rule just reduces to our old deterministic one.

If we let such a network run we can generate a probability distribution over states that it visits. We must be careful, however, to ensure that we measure this distribution only after the network has reached *thermal equilibrium* which simply means that the *averages* of the quantities we will be measuring to characterize our distribution (for example the average activation of the i_{th} unit $< S_i >$) are not changing over time. How do we know if the network will ever reach such an equilibrium ? Fortunately it turns out to be true that any network that always went downhill in some Lyapunov function in its deterministic version is guarenteed to reach a thermal equilibrium in its stochastic version. One important point to notice here is that the initial state of the network which was so crucial in the deterministic version is now *unimportant*, because if we run the network for long enough, the equilibrium probability distribution over states will be the same no matter which state we began in.

In deterministic Hopfield nets, we choose the weights as the sums of pattern correlations in order to make certain patterns stable points. But how do we set the weights in a stochastic network to make the distribution it generates match our world distribution ? At thermal equilibrium, the probability of finding the network in any particular state α depends only on the energy of that state, and is given by:

$$P_\alpha = \frac{e^{-E_\alpha}}{\sum_\beta e^{-E_\beta}}$$

where P_α is the equilibrium probability of being in state α, and the sum in the denominator is over all possible states. From this and our original energy equation, we can compute how the weights change the probabilities of the states:

$$\frac{\partial lnP_\alpha}{\partial w_{ij}} = \frac{1}{T}\left(S_i^\alpha S_j^\alpha - \sum_\beta P_\beta S_i^\beta S_j^\beta \right)$$

where S_k^γ is the state of the k^{th} unit in state γ. These derivatives can in principle be used to train the connection weights, however as we will see when we introduce Boltzmann machines, there is a better way to find the weights in stochastic networks.

2 Boltzmann Machines

We saw in our discussion of stochastic networks that we could design networks which always moved from state to state without settling down into a stable configuration by making individual neurons behave probabilistically. If we measured the fraction of the time that these networks spent in each of their states when they reached thermal equilibrium, we could use such networks to generate probability distributions over the various states. We further saw that we could encourage this equilibrium distribution to be similar to our world distribution by changing the connection weights in the network. However, our prescription for the derivatives of state probabilities with respect to the weights included only terms involving the activation states of *pairs* of units $S_i S_j$. This means that such networks will never be able to capture any structure in our world probability distribution that is higher than second order. For example, we could never train such a network with three units to visit the states $(0,0,0)$, $(0,1,1)$, $(1,0,1)$, and $(1,1,0)$ with some probabilities, but not the other four possible states. This is because the first and second order statistics (the mean of each component and the correleations between components) are the same for these four states as for the remaining four, and so the network cannot discriminate between them. This is a limitation of our stochastic networks (and indeed of all networks with no hidden units and only pairwise connections), and it comes from the fact that the state vectors which we are using as our patterns involve *every* unit's activation. If we were to use the activations of only a certain *subset* of units as our patterns then our networks would be able to capture higher order regularities in the distributions because some of the units would be free to represent these regularities. Such a scheme turns out to work; the units involved in the patterns are then called *visible* units, and the others *hidden* units.

Boltzmann machines are essentially an extension of simple stochastic associative networks to include hidden units – units not involved in the pattern vector. Hidden units, however, introduce a new complication: now knowing the world probability distribution of patterns which we want our network to reproduce tells us only what the probability distribution of the states of the *visible* units should be. We do not know what the probability distribution of the hidden units should be, and hence we do not know the full probability distributions P_β of the entire network state which we needed to calculate our weight derivatives $\partial lnP_\alpha / \partial w_{ij}$ so we can't train our connections. We could make up the hidden unit distributions somehow, but in fact the whole idea is that we would like the network itself to discover how to use the hidden units to best represent the structure of our distribution of patterns, and so we do not want to have to specify their probabilities. Clearly the old Hopfield net rule of $w_{ij} \propto s_i s_j$ will not help us with the wieghts to hidden units since we only have knowledge of s_i for visible units. How then will the connection weights get set ? Boltzmann machines provide a learning algorithm which adapts *all* the connections weights in the network given only the probability distribution over the visible units. Let us see how this works.

Consider a set of patterns α and the real world probability distribution P_α^+ over these patterns. For each component in these pattern vectors, we create a visible unit in the Boltzmann machine whose activity is associated with the value of that component. We also create some number of hidden units which are not part of the patterns that we hope will represent higher order regularities. *All* units in a Boltzmann machine compute an "energy gap":

$$\Delta E_i = E_{-1} - E_{+1} = \sum_j w_{ij} S_j$$

and then set their state according to the stochastic update rule:

$$p_i(+1) = \frac{1}{1 + e^{\Delta E_i / T}}$$

If we wait long enough, the system will reach a low temperature thermal equilibrium in which the probability of being in any global state depends only on its energy (divided by the temperature)[3]. We can estimate the probability distribution over the visible units in this "free-running" mode by sampling the average activities $< S_i >$ of all the visible units. Call this measured distribution P_α^- — we want it to be close to our desired distribution P_α^+. We will use the *Kullback Leibler distance*[4] between the distributions P^+ and P^- as a metric of how well our model is reflecting the world:

$$G = G(P_\alpha^+ \| P_\alpha^-) = \sum_\alpha P_\alpha^+ ln \left(\frac{P_\alpha^+}{P_\alpha^-} \right)$$

We can think of G as being similar to energy functions we have used for our networks in the past in that we would like to minimize it[5]. As such, we would like to know how changing various weights will affect G. This brings us to the Boltzmann machine learning procedure.

3 Learning

It turns out that all of the information about how a particular weight change alters the system energy G is available *locally* if we are willing to be patient enough. The learning procedure for the Boltzmann machine has two phases. In $Phase^+$, the visible units are *clamped* to the value of a particular pattern, and the network is allowed to reach low temperature thermal equilibrium. We then *increment* the weight between any two units which are *both* on. This is like Hebbian learning. This phase is repeated a large number of times, with each pattern α begin clamped with a frequency corresponding to the the world probability P_α^+ we would like to model. In $Phase^-$, we let the network run freely (no units clamped) and sample the activities of all the units. Once we have reached (perhaps by annealing) a low temperature equilibrium (and not before) we take enough samples to obtain reliable averages of $s_i s_j$. Then we *decrement* the weight between any two units which are *both* on. This is called *unlearning.* If we alternate between the phases with equal approximately equal frequency ($Phase^+$ should acutally be run a little more often), then this learning procedure will on average reduce the cross-entropy between the network's free-running distribution and our target distribution[6]. It amounts to saying that:

$$\frac{\partial G}{\partial w_{ij}} = -\frac{1}{T} (< s_i s_j >^+ - < s_i s_j >^-)$$

where $< s_i s_j >^+$ and $< s_i s_j >^-$ are the probabilities (at thermal equilibrium) of finding both the i^{th} and j^{th} units on together when the network is clamped and free-running respectively. For the visible units, $< s_i s_j >^+$ is set by the target distribution of patterns that we are clamping, but for the hidden units which are free in both phases, $< s_i s_j >^+$ will be whatever representation of higher order regularity the network chooses. The amazing thing about this rule is that it works for *any* pair of units, whether both visible, both hidden, or mixed. The equation makes intuitive sense for visible units: If $< s_i s_j >^+$ is bigger than $< s_i s_j >^-$ it means that units i and j are not on together in the free

[3] At high temperatures we reach equilibrium quickly, but low energy states are only slightly more likely than higher energy ones. At low temperatures, the probability of low energy states is significantly higher but it takes forever to get to equilibrium. A strategy known as *simulated annealing* which reduces the temperature as the network runs is a fast way to achieve a low temperature equilibrium.

[4] This measure $G(x\|y)$, also know as the *relative entropy* between two distributions tells us the inefficiency of assuming that y is the distribtion when the true distribution is x. In other words, if we knew x we could construct a code of average length $H(x)$ but if we only know y then the best code we can build has average length $H(x) + G(x\|y)$.

[5] Peter Brown (see PDP Chapter 7) has pointed out that minimizing G is equivalent to maximizing the log likelihood of generating the environmental probability distribution when the network is running freely and at equilibrium

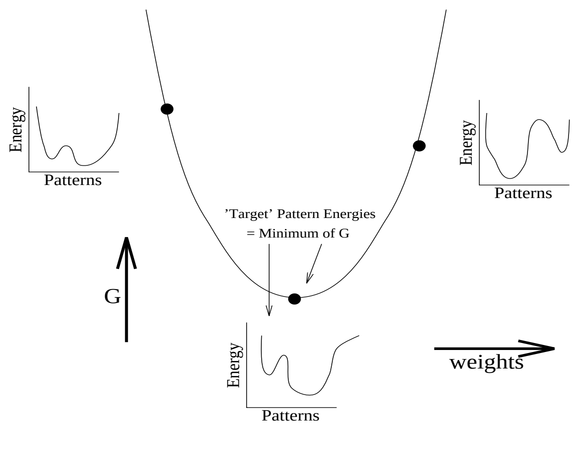
[6] For a proof, see Appendix A of Chapter 7 in PDP.

running phase as often as they should be (according to the clamped target distribution phase). So we would expect to want to *increase* w_{ij}, the connection between them; which is exactly what the equation says to do in order to reduce the energy G.

A crucial point that is often misunderstood is that the information about how a weight will change the energy G is only available locally from the *time evolution* of the activities. Only when we settle to equilibrium and take many samples there will this information "grow" out of the noise in the system. It is impossible to tell just from the activities of two units at one instant how changing the weight between them will affect G.

Let us try to understand what is going on in the above algorithm. During $Phase^{+}$, we are showing the machine what we want it to model and encouraging it to mimic our distribution by positive learning. During $Phase^{-}$, we are attempting to eliminate accidental or spurious correlations between units by simply degrading all pairs that are active together, under the assumption that the correct correlations will be built up again during the positive phase.

At the highest level, we are simply doing gradient descent in the function G. Each time we run a phase we are able to compute a small change in a single weight which will (on average) reduce G. But each point in G space (weight space) is actually a whole energy landscape over all the machine's possible states. So each time we make an adjustment to the weights to reduce G, the energies of all states change. In effect our goal in minimizing G is to deform the energy landscape so it matches our target energies over all the patterns. This is shown in the figure below:



For each step we wish to take on the surface, we must run all of our patterns, both in the clamped and unclamped phases. Only then do we obtain the information for one weight update.

Now, nested inside this top level search is another search that we must perform each time we want to compute a step in G (weight) space. We have to settle to a low temperature thermal equilibrium in the "current" energy landscape in order to achieve a state in which low energy states occur much more frequently than high energy ones. To do this, we use simulated annealing or some other search technique but this usually takes quite a bit of time. We can't skip this step, or else the probabilities P_{α}^{-} that we sample will not be representative of the current landscape's structure. Finally, nested within this settling is yet another time consuming process: once we are at thermal equilibrium we have to spend a large amount of time there sampling the activities of units in order to get a good estimate for $(< s_i s_j >^{+} - < s_i s_j >^{-})$[7].

4 Good and Bad Features

Boltzmann machines have been found to give excellent performance on many statistical decision tasks, greatly outstripping simple backprop networks[8]. Their ability to encode greater than second order regularities in data makes them extremely powerful. They also provide a very convenient Bayesian measure of how good a particular model or internal representation is – they simply ask: How likely is the model to generate the distribution of patterns from the world ? In this sense they incorporate the maximum likelihood principle directly into their structure. However, they are excruciatingly slow. This is in part due to the many nested loops involved in the learning procedure (as described above). But it is also largely due to the fact that Boltzmann machines represent probabilities *directly*: their units are actively turing on and off to represent a certain activity level, not simply holding a value which encodes that level of activation. There has been much research into an approximation of the Boltzmann machine dynamics known as the *mean field* approximation which attempts to address these bottlenecks. It degrades performance slightly but is found to be much faster; however that is a handout all to itself.

[7] Since we are taking the *difference* of two noisy random variables in order to estimate the correlation, the error only decreases as $\sqrt{N}$ for N samples.

[8] See in particular the study by Kohonen, Barna & Chrisley, *Statistical Pattern Recognition with Neural Networks* (1988) in IEEE ICNN San Diego, pp. 61-68 volume I.