9: Cubic nodes (contd.) and Reward Penalty training

Kevin Gurney

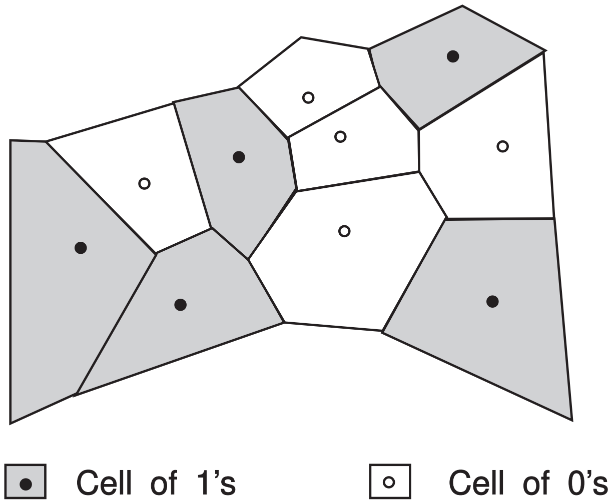
Dept. Human Sciences, Brunel University
Uxbridge, Middx. UK

This lecture deals with training nets of cubic nodes and introduces another major (quite general) algorithm - Reward Penalty. Insight into how we might train nets of cubic nodes is provided by considering the problems associated with generalisation in these nets. We then go on to consider feedback or recurrent nets from the point of view of their implementing iterated feedforward nets (recall this discussion in the case of Hopfield nets). Although the discussion here centres on cubic nodes, it provides insight into recurrent nets quite generally. Reward Penalty is introduced and is shown to apply to cubic as well as semilinear nodes.

1 Generalisation in Cubic Nodes - centres and clustering

First, recall the action of TLUs for comparison. The operation of an n-input TLU on Boolean vectors is determined by a hyperplane passing through the n-cube, so that all vectors on one side of this plane will produce a '1', while the others generate a '0'. Suppose a TLU has been trained to classify two input vectors. Every other possible input pattern will now be classified according to the node's hyperplane and there is automatic generalisation across the whole input space. (Recall the training of a 2-input TLU with only 2 vectors).

Consider now, a cubic node which, in the untrained state, has all sites set to zero. The output to any vector will be totally random with there being equal probability of a 1 or a 0. If this node is now trained on two (Boolean) vectors, only the two sites addressed by these will have their values altered; any other vector will produce a random output and there has been no generalisation. We shall call sites addressed by the training set *centre sites* or *centres*. In order to promote Hamming distance generalisation, sites close to the centres need to be trained to the same or similar value as the centres themselves. That is, there should be a *clustering* of site values around the centres. This is true in both feedforward and recurrent cube- based nets and there are various ways of achieving this which are discussed later. The situation is shown schematically in the diagram attached.



[slide of Voronoi tessellation]

Here, the extreme case is shown where the entire cube has been partitioned and no sites remain untrained. This algorithm for doing this has led to a *Voronoi* tessellation of cube, where the value a site takes on is determined by the value of its nearest centre; sites equidistant from opposite centres remain at 0. Training can now be seen as a two stage process; first centre sites have to be established according to the training set and then clusters developed around them. Before looking at ways of doing this, however, it is instructive to look more closely at the way clustering may help recurrent nets in auto-associative pattern recall. *

It is useful at this stage to summarise certain aspects of cubics nodes by making a comparison with the more usual semilinear weighted variety

- Cubic nodes allow greater functionality than semilinear nodes and therefore may be expected to implement some problems with fewer nodes.
- Cubic nodes have a ready implementation in RAM
- Cubic nodes do not exhibit intrinsic generalisation; they must be coerced to do so. Compare with linear dichotomy of TLUs
- The resources required for cubic nodes increases exponentially (number of site values varies like 2^n). May require use of MCUs.

2 Recurrent nets of cubic nodes

Although the discussion here centres on cubic nodes, it provides insight into recurrent nets quite generally.

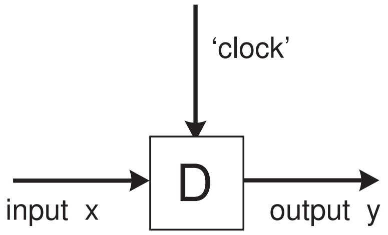
3 The Action of Well-Trained Recurrent Nets

We will limit the discussion here to the case where the unit output function is a hard limiter or very steep sigmoid so that any non-zero site value gives rise to a deterministic output. This

*Just as in the Hopfield nets in the 'A', 'B' simulation demo.

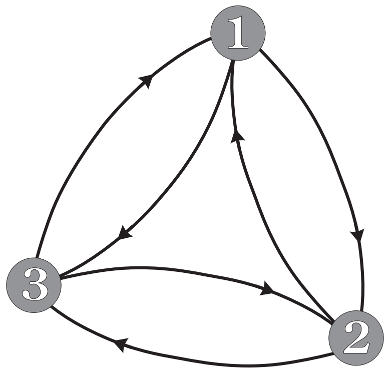
clarifies the argument which may later be qualified to take into account any probabilistic behaviour. The dynamics are also supposed, in the first instance, to be synchronous (all nodes update at the same time).

It is useful to think of a recurrent net performing pattern recall as iterating an input-output relationship. This is made explicit by 'unfolding' the net into a feedforward one, obtained by breaking the feedback loops and sending signals through a sequential iteration of hardware rather than through the same physical units. This was alluded to in the lectures on Hopfield nets but is restated here in the specific context of synchronous dynamics. A key component here is the use of delay or buffering elements (or temporary storage/latches) to avoid signal 'racing'. These are shown below



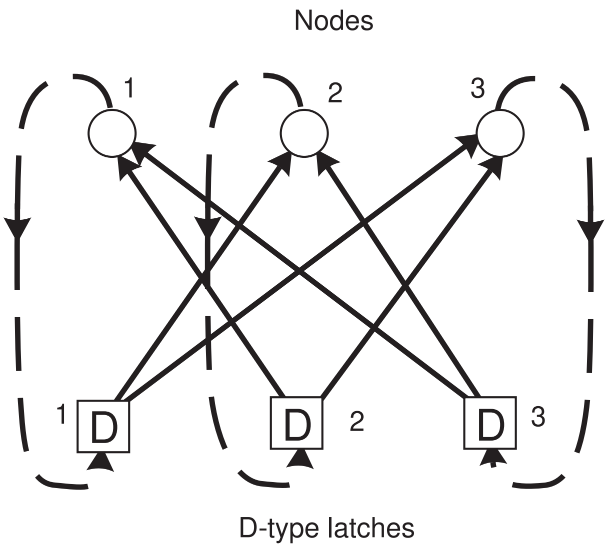
[D-type latch]

Their operation is very simple. Upon the application of a 'store' or 'clock' signal, the output of the device becomes equal to its input. Any alteration of the input has no affect on the output until another clocking signal is received. Now consider the (usual) 3 node network shown below.



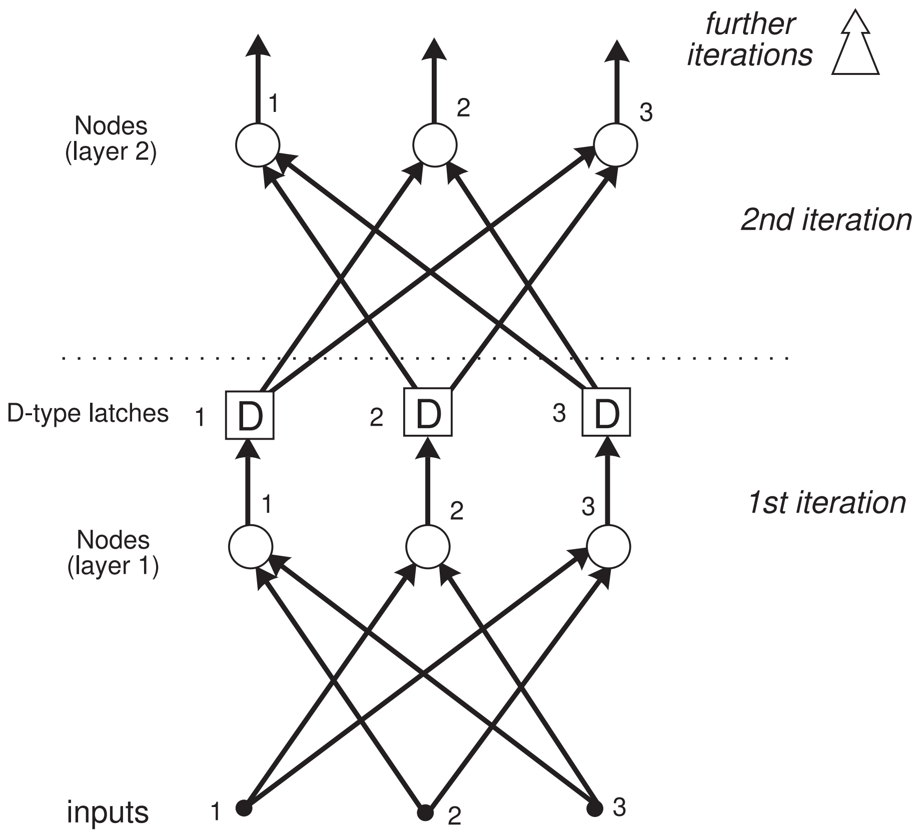
[Recurrent 3 node net]

In order to run this synchronously, we must store the previous state while evaluating the next state of the net. This may be done using a set of D-type temporary storage elements. These are not shown explicitly in the above representation but are shown in the network redrawn as a layered structure below.



[3 node net redrawn in layered form]

The state of the net is given by the contents of the Delay elements. To find the next state, the nodes process their inputs from the D-types, output the new values and then the D-type are 'clocked' to store the new state. This process is iterated indefinitely or until a single-state cycle has been reached. This iteration may be done using the same physical hardware, by feeding back the output as shown above, or by feeding the output forward into a replica of the same hardware as shown below.



[unfolded net]

The processing which takes place in the kth time step in the recurrent net now takes place at the kth layer of the feedforward net.

We now consider the processing of an initial state **u** ('input' in the unfolded net) which is close in Hamming distance to a trained state cycle **v**. Suppose the net has well clustered nodes; that is each centre is surrounded by a cluster of similar valued sites to the centre value itself. In the ideal case, starting from a pattern **u** close to a member **v** of the net's training set, the latter is recalled after one time step (single layer in the unfolded net). This happens because, although some of the sites addressed in some nodes are not centres, they are still sufficiently close to the desired centres to fall within the clusters associated with them and therefore give the same value as the centres themselves. In general, however, it will take several transitions (layers) to do this since some addressed sites will fall outside the relevant clusters. Thus, after one transition (layer), **u** will have been transformed into $\mathbf{u}'$, where the latter has more components in common with **v** than **u**. This has occurred because enough sites addressed by **u** are in the clusters associated with the centres of **v** and hence have a similar value. In subsequent transitions (layers) the same occurs but with fewer sites being visited which are outside the relevant clusters (or those of the correct value). The number of these erroneous sites should decrease to zero as the net moves through state-space.

A similar argument may be applied to the operation of recurrent nets of TLUs (e.g. Hopfield nets) but here the clusters are restricted to two regions of the n-cube which are linearly separable.

3.1 Training recurrent nets

3.1.1 Training centres

It is a simple matter to enforce a state cycle in a feedback network. We force or *clamp* the Delay latches to the desired state and train each node so that its output is just equal to the value on its corresponding latch. The sites addressed during training are, of course, centre sites. Note that there may be conflict between the training required for different vectors. This will occur when there is a component subset of a vector, forming the address to a node, which is the same as the corresponding subset of another vector, and the node is supposed to give opposite outputs in each case. For example, we couldn't train the 3-node net to the two states (001) and (000) since node 3 is required to give opposite outputs in each case to the address (00). To a lesser degree, the problem will become manifest if there are centres of opposite value which are close together. This leads to small clusters around these centres and, as the problem grows, the cube becomes 'fragmented'. These difficulties may be overcome in the following way.

Suppose that we add units to the net. It may now be possible to augment the original training vectors to include components on the new units so that centre value conflicts are removed and fragmentation reduced. Thus, in the 3-node net, adding a hidden node (labelled 4, say) which takes on different values for each of the conflict-producing vectors, will allow the two (augmented) vectors (001*0*) and (000*1*) to be learnt (the values of node 4 are in italics). These units are properly regarded as *hidden* since we are not interested in their vector components *per se* and may not impose these from outside; rather, the net should 'discover' them for itself. In terms of state-space, hidden units are being used to prevent overlap of centres of attraction and helping make each basin of attraction as large as possible.

This process of centre optimisation may be likened to a physical process where centres are allowed to move over the vertices of the cube and are subject to inter-centre forces. By making unlike centres repel we aim to avoid cube fragmentation and conflict. As in a physical system, the forces may be defined via a potential energy function. The problem then reduces to a function minimisation and may be tackled with simulated annealing. Further details may be found in (Gurney, 1989) together with simulation examples.

3.1.2 Developing site clusters

The most straightforward way to do this is to suppose that the net is endowed with enough processing ability to perform the Voronoi tessellation algorithmically. This will result in a net with site values $\pm S_m$ within cells and 0 at any sites on cell boundaries. This is an *off-line* technique in that it is not naturally done by the intrinsic neural hardware during training. Further details may be found in (Gurney, 1989; Gurney, 1992b). Aleksander (1991) has independently suggested a similar technique and embodied it in the so-called gRAM node.

Another possibility is to present noisy copies of the training set, thereby visiting sites close to true centres. This has been the basis of methods developed by Milligan (1988) who describes the process as one of modifying the net's state-space structure. It may be done naturally using an enhancement of the node's architecture described in (Gurney, 1992a).

4 Reward Penalty training

Consider a single node which has stochastic output. It may be semilinear or cubic, the only requirement is that it gives a boolean output which depends on the activation stochastically according to some nonlinear law like the sigmoid. Suppose that, just as for the delta rule, there is a set of input-output pairs so that each vector is associated with a 0 or 1. On applying a vector and noting the output, we compute a signal which is '1' ('Reward') if the node classified the input correctly, and '0' ('Penalty') if it classified incorrectly. If the node is rewarded, it adjusts its internal parameters (weights or site values) so that the current output is *more* likely with the current input. If, on the other hand, the node is penalised, it adjusts its parameters so that the current output is less likely. Specifically, for a stochastic semilinear node, the change in the weight Δw_i on input i

$$\Delta w_i = \begin{cases} \alpha[y - \sigma(a)]x_i & \text{if} \quad r = 1 \\ \lambda\alpha[1 - y - \sigma(a)]x_i & \text{if} \quad r = 0 \end{cases} \tag{1}$$

Here α is a learning rate, as usual, and λ is a constant which governs the relative importance of penalising. Empirically, λ values of around 0.2 appear most successful.

It can be shown (Williams, 1987) that this leads to a stochastic or noisy gradient descent. We are using less information than with the delta rule where we explicitly calculated the error gradient. This means that training will take longer. However, the training rule works for both hidden and output nodes directly without any modification. Recall that there was a substantial amount of calculation involved in evaluating the δ's in backpropagation. There is therefore a tradeoff between the complexity of each training step and the number of steps needed. Reward-Penalty (RP) also has the benefit that there is some noise which may be useful in evading entrapment in local minima. When there is more than one output node it is convenient to define an error e which is normalised to the range [0,1] and then define the probability of rewarding as $1 - e$

Barto and Jordan (1987) have successfully used RP in training nets of semilinear nodes. In fact, they train the output layer with the delta rule and the hidden layer with RP. This is sensible since the amount of computational overhead in the delta rule is small and it greatly increases the rate of learning overall. I (Gurney, 1989; Gurney, 1992a) showed that the above learning rule led to convergence for cubic nodes by adapting the proof of Williams (1987). Aleksander and Myers (1988) have developed an algorithm with an RP 'flavour' which they use to train nets of PLNs. Work is actively being pursued in developing hardware to train nets of MCUs using this algorithm (Hui et al., 1993; Bolouri et al., 1994) and the first generation of such chips has now been fabricated and tested (see poster in prefab 3).

References

Aleksander, I. (1991). (not sure of chapter title). In Eckmiller and Hartmann, editors, *Parallel Processing in Neural Systems and Computers,*, pages 2 – 5. North Holland.

Barto, A. and Jordan, M. (1987). Gradient following without backpropagation in layered networks. In *1st Int. Conference Neural Nets, San Diego*, volume 2.
(I have this).

Bolouri, H., Morgan, P., and Gurney, K. (1994). Design, manufacture and evaluation of a scalable high-performance neural system. *Electronics Letters*, 30:426.

Gurney, K. (1989). *Learning in networks of structured hypercubes*. PhD thesis, Dept. Electrical Engineering, Brunel University, Uxbridge, Middx, UK. Available as Technical Memorandum CN/R/144.

Gurney, K. (1992a). Training nets of hardware realisable sigma-pi units. *Neural Networks*, 5:289 – 303.

Gurney, K. (1992b). Training recurrent nets of hardware realisable sigma-pi units. *International Journal of Neural Systems*, 3:31 – 42.

Hui, T., Morgan, P., Gurney, K., and Bolouri, H. (1993). A cascadable 2048-neuron VLSI artificial neural network with on-board learning. In Aleksander and Taylor, editors, *Artificial Neural Networks 2*, pages 647 – 651. Elsevier.

Milligan, D. (1988). Annealing in ram-based learning networks. Technical Report CN/R/142, Dept. Electrical Engineering, Brunel University.

Myers, C. and Aleksander, I. (1988). Learning algorithms for probabilistic neural nets. In *1st INNS Annual Meeting*.

Williams, R. (1987). Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3, Northeastern University Boston.