Neuron Splitting for Efficient Feature Map Formation

Lachlan L. H. Andrew
Department of Electrical and Electronic Engineering
University of Melbourne, Parkville, Victoria 3052, Australia
lha@ee.mu.OZ.AU

***Abstract*—Kohonen's Self Organising Feature Map (SOFM) produces an ordered mapping from one space to another. This paper describes an algorithm inspired by the splitting initialisation for the classical LBG method for vector quantiser design, which allows the efficient generation of maps with various topologies and with high local and global ordering.**

I. Introduction

Kohonen's Self Organising Feature Map (SOFM) [1] is a biologically plausible artificial neural network modeling the formation of an ordered mapping between specific concepts and locations in the brain. The resulting map is ordered, or topology preserving, and also reflects the distribution of its inputs in the sense that regions of the input space with a high probability density are allocated more output values (neurons) than regions of low probability. This has proved to be useful in many applications [2–5]. Another algorithm which also provides a mapping with some useful ordering is the LBG or generalised Lloyd algorithm [6] using the splitting initialisation. By forming a global structure on smaller maps and then splitting them once the order has formed, this algorithm leads to efficient implementations on serial computers, but it is limited to producing a single topology. Although this topology is useful for tree search vector quantisers [7] and progressive transmission [3], it is but one of the many useful topologies. This paper presents an algorithm combining the efficiency of the LBG algorithm with the generality of the SOFM algorithm, by allowing a wide range of topologies to be generated using splitting.

II. Summary of the Existing Algorithms

A. SOFM training

Kohonen's SOFM algorithm is a type of soft competitive learning. When an input vector $\boldsymbol{x}$ is presented to the network, the neuron, r, whose weight vector, $\boldsymbol{w}_r$, is closest to $\boldsymbol{x}$, is selected as the winner. All neurons are then updated according to the rule

$$\boldsymbol{w}_s(t+1) = \boldsymbol{w}_s(t) + \eta(t) h_{rs}(t)(\boldsymbol{x} - \boldsymbol{w}_s(t)),$$

where $\boldsymbol{w}_s$ is the weight vector of neuron s, η is the learning rate, and h_{rs} is the neighbourhood function. For each neuron r there is a vector $\boldsymbol{r}$ in the output space, which is typically one or two dimensional, and the neighbourhood function is normally a decreasing function of the distance in output space between its two arguments:

$$h_{rs}(t) = h(\|\boldsymbol{r} - \boldsymbol{s}\|, t), \tag{1}$$

where h is a monotonically decreasing function. It is the updating of neighbours, governed by the neighbourhood function, that imparts the structure on the codebook by causing the weights of neurons which are close in output space also to be close in input space, since they are updated from the same input vector $\boldsymbol{x}$. It is also responsible for making sure that all neurons represent some part of the input space, since neurons outside the convex hull of the input space are gradually brought towards it when their neighbours win.

To ensure a good global ordering of the map, it is necessary that, in the early stages, the neighbourhood function cover many of the neurons. In detailed simulations, the neighbourhood is often chosen to be a function of infinite support such as a Gaussian, whose variance decreases as training proceeds. However, many simulations neglect the weak long range interactions in later stages and simply use a piecewise constant neighbourhood

$$h(d,t) = \begin{cases} 1 & \text{if } d \le \theta(t) \\ 0 & \text{otherwise} \end{cases}$$

where $h(d,t)$ is the function in (1) and $\theta(t)$ is a threshold which decreases as training proceeds. It has been shown [3] that the LBG algorithm with splitting described below is of this form, when the neurons are arranged appropriately in output space.

B. LBG training

The LBG algorithm is a batch training algorithm which generalises Lloyd's algorithm for scalar quan-

tiser design [8]. Without the splitting component, it consists simply of repeatedly applying two steps:

LBG1 Partition all the training vectors to be used in this epoch such that each is assigned to the neuron with the closest weight vector.

LBG2 Adjust the weight vector of each neuron to the centroid of the training vectors assigned to it.

The splitting algorithms initially trains a network of only one neuron. When this has trained sufficiently that further training produces negligible change, a copy of this neuron is created, with a small random offset to enable a meaningful partition to be formed in the next LBG1 stage. When these two have trained sufficiently, each neuron is again split and the process is repeated until the network of the required size is created. When training has finished, the resulting mapping has some structure since two neurons which split early must spread far apart to represent the input adequately, while neurons which split late need not change much. This structure is enhanced if at each stage, each of the new neurons has the same offset added. If neurons are numbered such that the ith bit is set if the neuron is derived from one created during the ith split, then the neuron numbering reflects the structure.

If the SOFM is implemented on a serial computer, the cost of selecting the winner increases with network size. This increase is typically linear, although fast optimal methods exist [9], and many suboptimal fast searches exist. If a given proportion of the neurons are to be covered by the neighbourhood function in the early stages of training, then the update cost is also linear in network size. Thus substantial computational savings can be made by using the splitting approach to reduce the size of the network in the initial stages.

III. Splitting Arbitrary Topologies

Since the LBG algorithm provides no means for enforcing order on the mapping between splits, it is limited to implementing one topology. An increased range of topologies can be generated by including explicit local interactions at each stage, as the SOFM does by means of its neighbour updates. Any topology can then be generated as long as the new neurons introduced by the splitting can be fitted into the lattice without disrupting its structure. Since the expansion of the network replaces a shrinking neighbourhood, only the winner and its nearest neighbours need to be updated ($\theta(t) = 1$), further reducing computation from that of the SOFM. Care must be taken that the splitting process does not disturb the order introduced by the SOFM, or equivalently, care must be taken that the two sources of order cooperate rather than competing. Below are descriptions of how the splitting algorithm may be combined with some common topologies.

A. Linear Chain

No special care is required to generate a linear topology, since that is very close to the topology already generated by the splitting algorithm. Indeed, this linear splitting is already in common usage (see for example [10]). Splitting is achieved by the assignment

$$\hat{\boldsymbol{w}}_r = \boldsymbol{w}_{r/2},$$

where $\hat{\boldsymbol{w}}_r$ denotes the new neuron weight, $\boldsymbol{w}_r$ denotes the old weight, and / denotes integer division with truncation. This inserts the new nodes next to the nodes from which they split. A final network whose size, m, is not a power of two may be generated by replacing the final split with one of the form

$$\hat{\boldsymbol{w}}_r = \boldsymbol{w}_{(r2^n)/m}, \qquad (2)$$

where 2^n is the size of the network before splitting. If the final size, m, has a factor which is a power of two then the partial split can be performed before the final stage, allowing more time for the network to smooth out the non-uniformity caused by the uneven split. Clearly the same strategy can be used to split a closed loop, changing only the neighbourhood function.

B. Rectangular Lattice

One of the most useful topologies [3] is a rectangular lattice on either a flat sheet or a torus. In order to form the two dimensional mapping, the splitting must occur in each dimension of the output space. This could be achieved by splitting each neuron into four, but a simpler way is to split each dimension alternately. Thus the mappings have dimensions 1×1, 2×1, 2×2, 4×2, and so on up to the desired size. Thus splitting alternates between

$$\hat{\boldsymbol{w}}_{r_1 r_2} = \boldsymbol{w}_{r_1/2, r_2}$$

and

$$\hat{\boldsymbol{w}}_{r_1 r_2} = \boldsymbol{w}_{r_1, r_2/2}.$$

This generates networks whose final size is a power of two and whose sides are in the ratio 1:1 or 2:1. Networks whose sides are not powers of two, but which are in a ratio between 1:1 and 2:1 may easily be generated by applying a partial split analogous to (2) on each of the last two splits, while ratios n:1, $n > 2$ can be achieved by splitting one way more often than the

other. Again this splitting can easily be used to split a cylinder or torus by adjusting the neighbourhood function appropriately.

C. Other Topologies

In general, n dimensional splitting of a hyperrectangular lattice can be achieved by cycling through the dimensions, splitting one at a time. In the case of the hypercube topology [4], each dimension only gets split once before the final size is reached. Thus extra training after the final split is required in order to make sure that each dimension is fully exploited.

Although it is less straightforward, a hexagonal lattice can be split in several ways by splitting the network as though it were a rectangular lattice, but using the explicit interactions of a hexagonal network.

IV. Direction of Splitting

In the original LBG algorithm [6], splitting was achieved by simply adding a random vector to each duplicate neuron. This is a simple but clearly suboptimal method. It may not provide a split in a useful direction, requiring more iterations to cause the weight vectors to represent the true clusters, or in the case of an ordered map, it may disturb the order of the network. A solution to the first problem has been proposed [11], based on splitting the neuron in the direction of greatest variance. This is an effective solution to the first problem, at the expense of a small amount of work to calculate the appropriate direction. A simple solution to the second problem is to split in the direction of the line joining the neuron to its neighbour. If the probability density is locally approximately uniform, as is the case with smooth clusters or large output maps, then the optimal position for the new neuron is the midpoint between the two neurons which will be on either side of it in the new lattice (midpoint splitting). If the distribution is not known to be approximately uniform then the new neuron can be formed close to the existing neuron, but still in the same direction. This is the direction which can be expected to be optimal. It is shown experimentally that even with no explicit interaction between neurons this can produce substantial ordering.

V. Results

Results of computer simulations are shown in figure 1. These are the results of training a 16×16 node flat network with random data uniformly distributed on a square using batch updates. Each of the 80 batches consisted of presenting 256 random vectors, and the network was split after every eight batches until its full size was reached. In trials with $\theta = 0$, only the winner was updated, and when $\theta = 1$ a neighbourhood of eight neurons was also updated with a constant interaction of $h(1,t) = 0.1$. In both cases the learning rate for the ith batch was $\eta(i) = 2/(2+i)$.

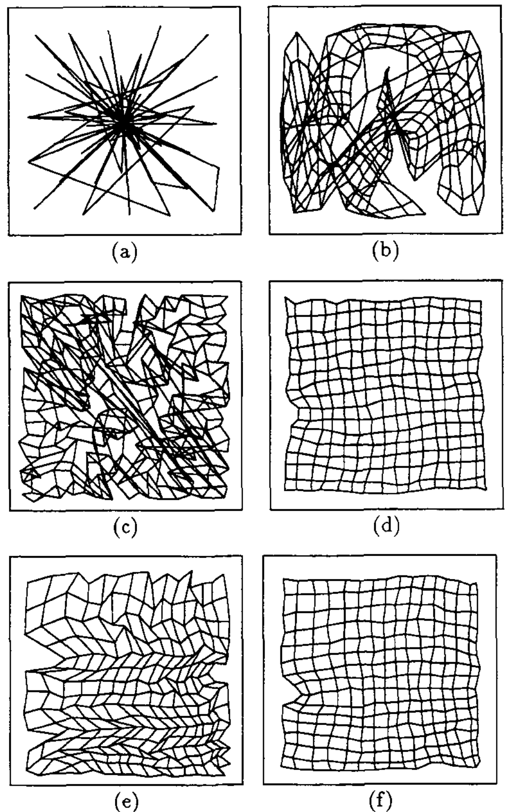

Fig. 1. Results of networks trained with neighbourhood radius θ and various splitting strategies. (a) $\theta = 0$, no splits (b) $\theta = 1$, no splits (c) $\theta = 0$, random splits (d) $\theta = 1$, random splits (e) $\theta = 0$, midpoint splits (f) $\theta = 1$, midpoint splits

Clearly the map in fig. 1(a) suffers from severe underuse (many of the vectors have not been updated at all) and is quite unacceptable. Local interactions without splitting cause a map with fair local order but no global order (fig. 1(b)). Even the least ordered of those using splitting, that using random splitting and no explicit interaction (fig. 1(c)), provides a detectable degree of global ordering. Midpoint splitting clearly increases the local order considerably, while neurons with explicit local interactions as well

as splitting form highly ordered maps (fig. 1(d),(f)). Greater order could have been achieved using a larger number of vectors per batch to obtain greater averaging of the random process, or by increasing the explicit interactions, but parameters have been selected which highlight the effect of splitting, and the results suffice to demonstrate the superiority of nets employing splitting over those only using explicit local interactions between neurons.

VI. Conclusion

It has been shown that feature maps with good global and local ordering can be produced efficiently using only local interactions between neurons by initially training small maps and then splitting them to form larger ones. By using an appropriate splitting procedure, different topologies can be generated by the splitting alone, even without explicit interactions.

VII. Acknowledgment

The author would like to thank his supervisor, Dr M. Palaniswami, for his comments on drafts of this manuscript. While performing this work, the author was on a scholarship from the Australian Telecommunications and Electronics Research Board (ATERB).

References

[1] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biol. Cybern.*, vol. 43, no. 1, pp. 59–69, 1982.

[2] L. L. H. Andrew and M. Palaniswami, "A study on the effect of neighbourhood functions for noise robust image vector quantisation," in *Proc. ICNN*, (Orlando, FL), pp. 4159–4163, 1994.

[3] L. L. H. Andrew and M. Palaniswami, "On the effect of neighbourhood functions for image vector quantisers," Submitted to *IEEE Trans. Neural Networks*.

[4] D. S. Bradburn, "Reducing transmission error effects using a self-organizing network," in *Proc. IJCNN*, (Washington, DC), pp. II-531–II-537, 1989.

[5] K. K. Truong, "Multilayer Kohonen image codebooks with a logarithmic search complexity," in *ICASSP 91*, (Totonto, Canada), pp. 2789–2792, 1991.

[6] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, vol. 28, pp. 84–95, Jan. 1980.

[7] A. Buzo, A. H. Gray, Jr, and R. M. Gray, "Speech coding based upon vector quantzation," *IEEE Trans. Accoust. Speech and Sig. Proc.*, vol. 28, pp. 562–574, Oct. 1980.

[8] S. P. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inform. Theory*, vol. 28, pp. 129–137, March 1982.

[9] C.-M. Huang, Q. Bi, G. S. Stiles, and R. W. Harris, "Fast full search equivalent encoding algorithms for image compression using vector quantization," *IEEE Trans. Image Processing*, vol. 1, pp. 413–416, July 1992.

[10] J. V. Black, "Comparison of the performance of vector quantiser training algorithms," in *Proc. ICANN*, (Brighton, UK), pp. 71–75, 25–27 May, 1993.

[11] C.-M. Huang and R. W. Harris, "A comparison of several vector quantization codebook generation approaches," *IEEE Trans. Image Processing*, vol. 2, pp. 108–112, Jan. 1993.