a)

b)

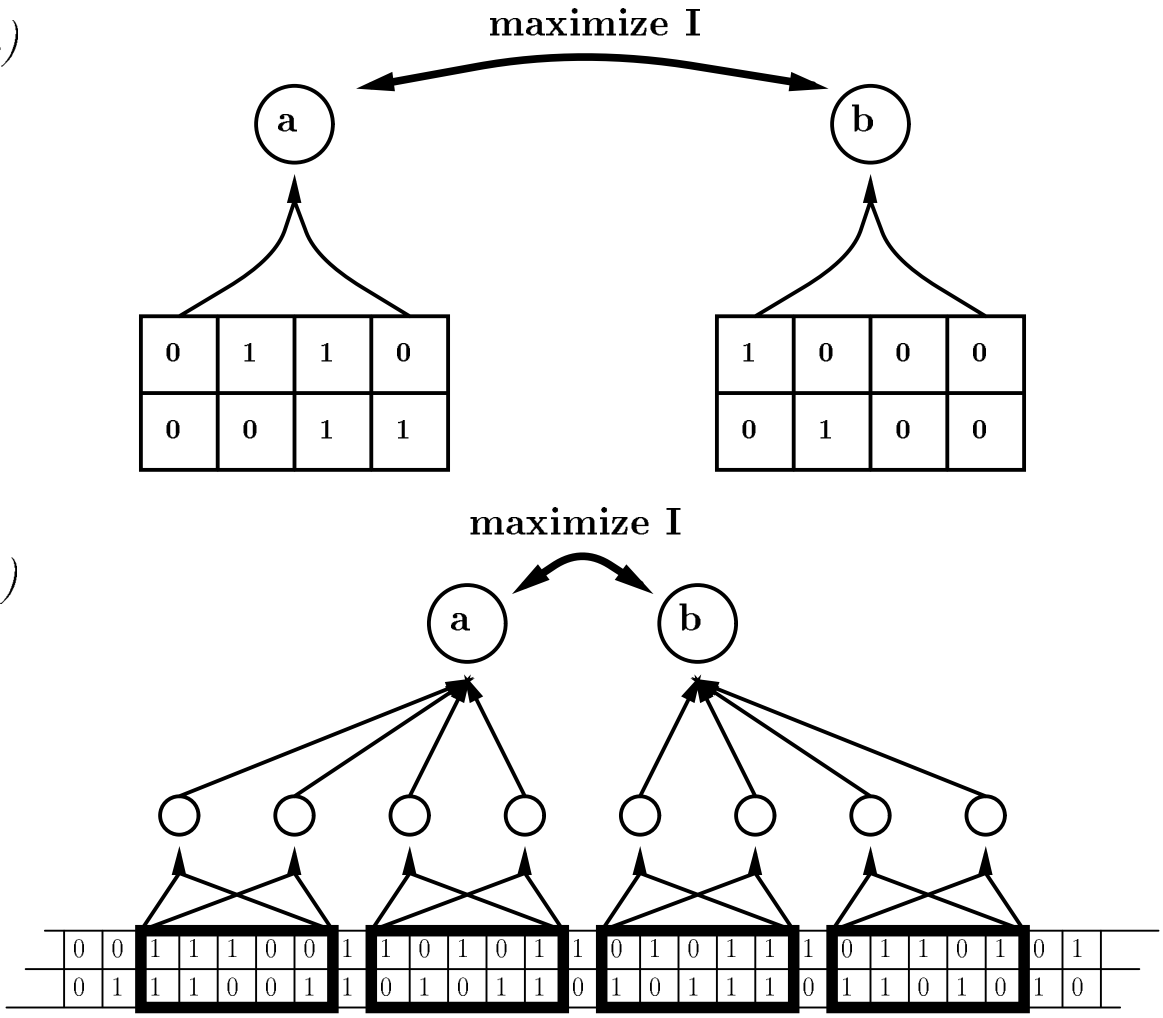

Figure 3.7: *a) Two units receive as input random binary patterns in which the left half is a shifted version of the right half. The inputs to the two units are unrelated apart from having the same shift. The learning algorithm adjusts the weights of each unit to maximize the mutual information, over the ensemble of training cases, between the states of the two units. b) In a multi-layer version of this architecture, once the first layer of weights has been trained to extract some low order features, the next layer can hierarchically combine these noisy estimates into more accurate feature estimates.*

receptive fields with a gap of one pixel between patches(see figure 3.7b).[2] The receptive field width n varied in different experiments. As before, each stochastic binary unit used the logistic nonlinearity to determine the probability of outputting a 1, and we used the exact value of this probability (rather than stochastic sampling) in our simulations. Units learned roughly by the method of gradient ascent, with the modification that the size of the weight change for each weight was truncated to lie within $[-0.1, 0.1]$. We found that in practice the gradients tended to increase by many orders of magnitude as the algorithm neared a solution, and this upper limit on the step size prevented the learning from "blowing up" in this situation.

We performed a number of preliminary experiments with the algorithm, varying the architecture and the training set size. With random patterns and a small training set there is a high probability that units will learn some of the random structure in the data in addition to the shift; as the number of training cases increases, sampling error decreases and units become more tuned to shift.

We can further increase the likelihood that shift will be learned, rather than random structure, by increasing the number of receptive fields (and hence the input size), because shift is the only feature common to the inputs of all units. When we extend the architecture shown in figure 3.7a to multiple receptive fields (with one unit receiving input from each receptive field) each unit now tries to maximize the sum of its pairwise mutual information with each of the other units. In this case, an interesting effect occurs as the learning proceeds: once some pair or subset of units within a layer has "caught on" to the shift feature, their mutual information gradients become very large, and convergence accelerates. This provides stronger shift-tuned signals to the other units, so that the effect rapidly spreads across the layer.

Since shift is a higher order feature of the input (i.e., it can be predicted from the products of pairs of pixel intensities but not by any linear combination of the individual intensities), a network with a single layer of weights (assuming units of the type defined in equation 1.3) cannot learn to become a perfect shift detector.[3] Hence, the problem requires a multilayer (nonlinear) network, so that with each successive layer, units can combine the responses of several lower level feature detectors to make more global predictions. Using a hierarchical architecture as shown in figure 3.7b, the partially shift-tuned noisy features extracted by the first layer from several patches of input can be combined by the second layer units, thereby increasing the signal to noise ratio in the prediction of shift. At the top layer, every "output" unit maximizes the sum of its mutual information with each of the others. In the middle layer, there are m clusters of units, each cluster receiving input from a different receptive field, and each output unit receiving input from a different set of clusters.

As mentioned above, the performance measure we use is the mutual information between the output units' activities and the shift, when shift is treated as a binary feature of the input pattern. For a network with two output units and four 2-unit clusters in the middle layer each with 2 by 5 receptive fields (as shown in figure 3.7b), the learning converged after about 300 passes through a training set of 500 patterns, and the output units learned to convey, on average (over 5 repetitions from different initial random weights), about .03 bits of information about the shift. As we increased the number of receptive fields, the ability of the network to learn shift increased dramatically. The most shift-tuned (and largest) network we experimented

[2] With no gap between the receptive fields of two neighboring units, those units could simply learn the uninformative correlations between the upper (lower) rightmost bit of the left unit's field and the lower (upper) leftmost bit of the right unit's field, while ignoring the rest of their inputs.

[3] See Minsky and Papert (1969) for a thorough treatment of the issue of what can and cannot be learned by single layer networks.

with on random shift patterns had 5 output units, each receiving input from two 2-unit clusters of units in the middle layer, and ten 2 by 4 receptive fields in the input layer. For this network, the learning took about 500 passes to converge on a training set of 1000 random patterns. The output units conveyed on average about .23 bits of information about the shift, and the most shift-tuned unit (over five runs from different initial random weights) conveyed .46 bits of information about the shift. Note that the maximum possible information that could be conveyed about a binary feature is 1 bit.

3.3.1 Using back-propagation to train the hidden layers

The multi-layer algorithm described above has the desirable property that layers can be trained sequentially. Once the initial layer learned partially shift-tuned feature detectors, subsequent layers can take advantage of this earlier learning and quickly become even more shift-tuned. However, there is no guarantee that the shift patterns will be perfectly classified by the top layer, since units in lower layers within a cluster may learn redundant features, which only apply to a subset of the input patterns. Since we only maximized the mutual information between *pairs* of hidden units, there was nothing to prevent many pairs from learning exactly the same feature. We can improve the performance of the network by applying our objective function between units at the top layer *only*, and using back-propagated gradients to train the middle layer. The back-propagated gradients send a more global training signal to the hidden units, which is best optimized by having hidden units differentiate to detect different features. In practice, as we shall see in the next subsection, a combination of the sequential, layer by layer, application of the same objective (as described in the previous subsection) and "top-down" or back-propagation training of the hidden units, seems to result in optimal performance.

We can reduce the possibility that units may learn any of the random input structure by creating a complete set of unambiguous binary patterns. Using 2 by 4 receptive fields, there are $2^4 = 16$ possible left-shifted and 16 right-shifted patterns within a receptive field. We remove the 8 ambiguous patterns[4]. We used two modules, and created 16 bit patterns by presenting each possible combination of pairs of the 12 unambiguous 2 by 4 bit left-shifted patterns, and each combination of pairs of right-shifted patterns, yielding a total training set of 288 16-bit patterns.

The hierarchical architecture shown in figure 3.7b was helpful in dealing with noise when we trained layers sequentially, because the output units could compute a weighted average of information from multiple receptive fields. However, when we are back-propagating derivatives to hidden units, we can deal with this problem by simply adding more hidden units for each receptive field, because the back-propagated signal from an output unit provides pressure for the hidden units to extract different features. In our experiments on the unambiguous pattern set (and in the remaining experiments reported in this section), instead of the hierarchical architecture used earlier, we used a modular architecture as shown in figure 3.8. Each 2 by 4 patch was used as input to a separate module that contained a layer of hidden units and one "output" unit.

Since we had removed the random aspect of the input patterns, we found we could get by with a much smaller network having only two modules. When we trained our two module network on this complete set of patterns (with back-propagation to the hidden layers), we usually got (at least one of two) output units

[4] The ambiguous patterns have 1111, 0000, 1010 or 0101 in the top half. Note that the left- and right-shifted versions of each of these (using shift with wrap-around within receptive fields) are indistinguishable. The use of wrap-around within each receptive field makes this example differ slightly from a simple version of a stereo task.

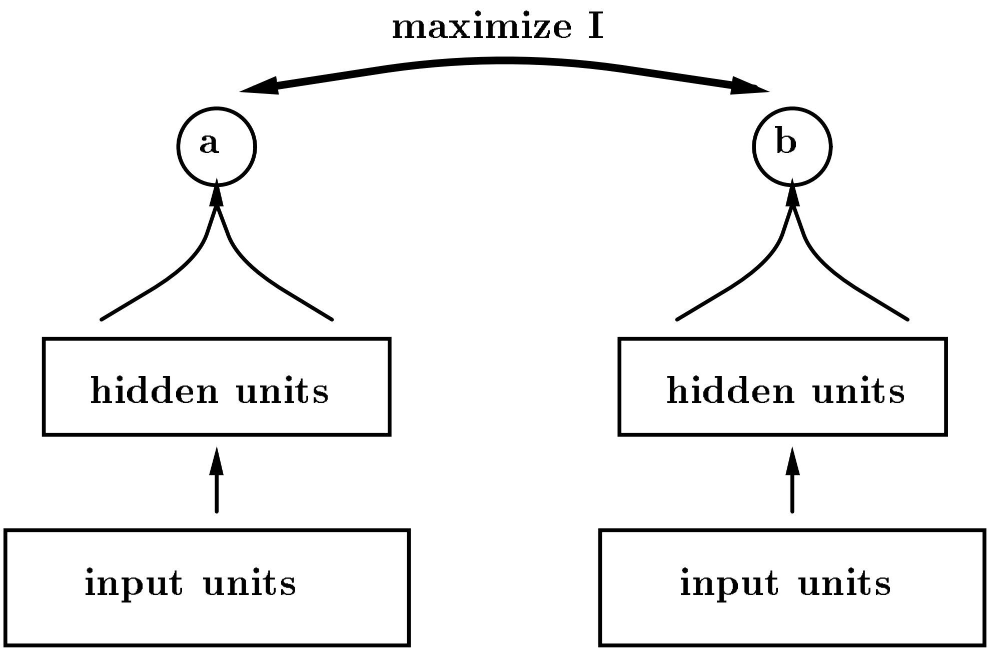

Figure 3.8: *Two modules that receive input from adjacent receptive fields. Each module has one layer of hidden units. The learning algorithm maximizes the mutual information between the states of the two output units, and the gradients are back-propagated to hidden units.*

that were pure shift detectors within about 300 passes through the training set. The mutual information between the two output units occasionally approached the global maximum of 1 bit (corresponding also to 1 bit of information about the shift), although usually the network found only locally maximum solutions, as shown in figure 3.9c.

3.3.2 Improving the performance with bootstrapping

The learning is rather slow for two reasons. First, we are not specifying desired values for the "output" units—we are only specifying that their pairwise mutual information should be high. The derivative of this objective function w.r.t a weight depends on *three* other layers of adaptive weights—one other layer in the same module and two layers in the adjacent module. So in this respect the difficulty of learning resembles back-propagation through four layers of weights. Second, with random starting weights, the initial gradient of the objective function is very small.

The performance of the algorithm, in terms of both convergence speed and quality of the final solution, is greatly improved by using a "bootstrapping" method. We start by applying our objective function between pairs of units within layers (as we did when in the hierarchical version) until these units are somewhat tuned to the shift. Then the gradients of the mutual information between the output units are much bigger and the objective function can be applied at that layer *only* and the derivatives back-propagated.[5] More globally coherent information can now be provided to the hidden units that failed to find any useful features in the bootstrapping phase.

[5] One could alternatively apply both objectives simultaneously to the hidden units.

We ran simulations with three versions of the algorithm, one with bootstrapping, one with back-propagation, and one combining the two training methods sequentially. We compared their performance on a network with two modules, using 8 hidden units and one output unit per module, on the complete, unambiguous pattern set described above. The three versions of the learning algorithm we used were:

Version 1. 50 bootstrapping learning iterations, in which each layer was trained to maximize information between pairs of units between modules, followed by 250 iterations with information maximization between the top two units and back-propagation of gradients to the hidden units.

Version 2. 300 iterations in which each layer was trained to maximize information between pairs of units in different modules in the same layer, without back-propagation (as in the bootstrapping phase of Version 1).

Version 3. 300 iterations with information maximization between the top two units and back-propagation to the hidden units (as in the back-propagation phase of Version 1).

Version 1 performed by far the best: in 48 out of 50 runs from different initial random weights, the network learned an exact solution to the shift problem, in which one or both output units predicted shift almost perfectly.[6] In Versions 2 and 3, while the top-level units nearly always became highly shift-tuned, only in 27 out of 50 and 30 out of 50 repetitions, respectively, did the network learn to perfectly separate the two classes of patterns. Figure 3.9 shows the learning curves for ten runs of the three versions; in some cases the mutual information between the top two units asymptotically approaches the global optimum of 1 bit, while in other cases the procedure has apparently become stuck in sub-optimal regions of weight space. Note that some of the curves have small oscillations; this is due to the optimization method used – steepest ascent with a maximum absolute weight change. When the gradients become very large, the procedure follows the gradient less closely, and therefore may not always climb uphill in mutual information.

It appears that the bootstrapping procedure increases the likelihood that the network will converge to globally optimal solutions. The reason for this may be that because we are applying two different learning procedures to the hidden units, they have more opportunities to discover useful features at each stage. During the bootstrapping stage, a given pair of hidden units attempting to achieve high mutual information cannot learn to predict shift perfectly, since the shift problem is not learnable by a single layer of weights. However, some units do learn to be partially correlated with shift at the bootstrapping stage. This makes the job of a pair of output units in adjacent modules much easier, since they now have more spatially correlated inputs, which are at least partially shift-tuned. Now when we back-propagate the information gradients to the hidden units, those hidden units which failed to learn anything useful in the bootstrapping stage are under increased pressure to become shift-tuned.

Not only does the bootstrapping procedure increase the overall quality of the solution, but on more difficult learning problems it substantially reduces the learning time. We find that for the continuous version of our algorithm (described in the next chapter), on stereograms with continuously varying depths, when the hidden units are initially trained with bootstrapping, the subsequent learning with back-propagation of gradients is accelerated by roughly an order of magnitude.

[6] I.e., the mutual information between the unit's output and shift was greater than or equal to 0.6 nats = about .86 bits. In the remaining two runs, the output units were still both highly shift-tuned, but did not meet this criterion.

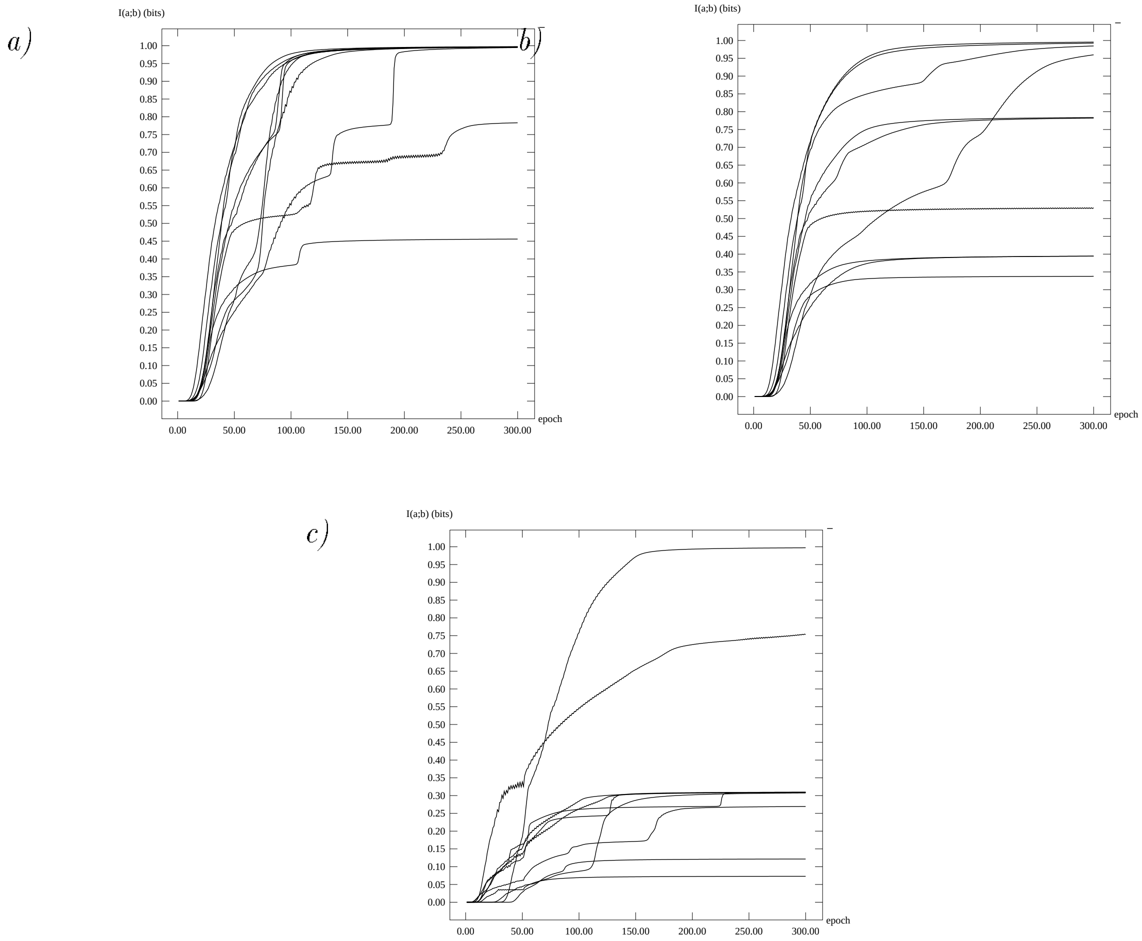

Figure 3.9: *Mutual information between two modules' outputs versus learning epochs on 288 binary shift patterns. a) shows learning curves for the combined case (version 1), using back-propagation preceded by 50 "bootstrapping" learning iterations for the hidden layer, for 10 runs starting from different initial random weights. b) shows learning curves using purely "bottom-up" learning (version 2 – training the network sequentially, one layer at a time), and c) shows learning curves using back-propagation alone (version 3). The learning curves in b) and c) were generated starting from the same initial weights as in a).*

3.4 Discussion

We have applied the binary Imax algorithm to two problems: learning to represent spatial frequency and phase in simple, sinusoidal intensity images, and learning to detect shift in binary stereograms. The n-valued version of discrete Imax was applied to pattern sets containing many frequencies, and was able to divide up the phase-frequency space among a group of units. The same extension could be applied to the stereo problem, to learn to represent multiple shifts. In general, this extension works well for the representation of a single parameter or set of parameters as an interval code, in which the parameter space is divided into a number of disjoint regions, and one unit is allocated to each region. One major drawback of n-valued Imax is that it is much slower than the binary version, since the number of probability statistics and gradient terms that must be computed increases as n^2. However, this effect is only significant for small networks in which the number of weights in the network is much less than n^2.

A second drawback of the version of the Imax algorithm presented in this chapter is that it can only represent one or more spatially coherent features as a single parameter, even if there are multiple coherent features in the data. It cannot detect multiple statistically independent features and represent them separately, as it only performs information maximization between two single variables (binary or n-valued). Thus, the phase and frequency parameters, which are actually statistically independent in the training sets used, are treated as a single joint variable. Another way to extend Imax is to maximize the mutual information between two *sets* of discrete variables, where each variable in each set could be binary or n-valued. Unfortunately, the complexity becomes even worse than for the case of Imax between two n-valued variables. The mutual information between two vectors is much more complicated, because all the higher order statistics between all possible subsets of the two variables must be taken into account. In the next chapter, we explore a version of Imax for continuous parameters, and discuss more tractable ways of extending this model to Imax between vectors.

A final drawback of discrete Imax is that units tend to adopt rigidly binary states on each case. This allows the network to do well with respect to the information measure, since a pair of units can strongly agree (or disagree) if their outputs are very close to zero or one. To achieve these extreme output values, however, the network must develop very large weights. As the magnitudes of the weights increase, the unit's probabilistic response function sharpens into a step function. On each particular case, the unit's response is at one of the extremes of this function, where its slope is virtually zero. This means that effectively the unit can neither adapt any further, nor can its output be used to indicate its degree of uncertainty about its decision, since it has virtually become a linear threshold unit. These problems are addressed by the continuous version of Imax, presented in the next chapter.

Chapter 4

Continuous Imax

In the previous chapter we proposed an objective for unsupervised learning, based on the assumption of coherence across different parts of the sensory input. We presented one algorithm for such a learning procedure, based on a discrete model of spatially coherent parameters in images. In this chapter, we show how to extend the algorithm to the continuous case (Becker and Hinton, 1989, 1992).

4.1 Modules with real-valued outputs

The discrete version of the Imax algorithm presented in the previous chapter has a number of drawbacks. In order to represent a wide range of values of a real-valued parameter with sufficient precision, there must be enough units to represent each possible parameter value of interest. Unfortunately, as the number of units gets large, the time complexity to compute the statistics for the mutual information derivatives grows as the square of the number of discrete values represented.[1] Further, the algorithm drives units to become extremely binary, acting virtually like linear threshold units. This binary behaviour makes it difficult to represent a continuous parameter such as depth in more realistic images that contain smoothly curved surfaces which have real-valued depths. In this case, we would like units to learn something like analog codes for depth.

The expression for the mutual information between two real-valued variables a and b is

$$I_{a;b} = -\left[\int p(a)\log p(a) + \int p(b)\log p(b) - \int\int p(a,b)\log p(a,b)\right] \tag{4.1}$$

To extend our algorithm to real-valued, continuous parameters, we must make some simplifying assumptions about the probability distributions of the variables, in order to obtain a tractable approximation to this expression. We start by making the following very simple coherence assumption (which will be relaxed later): There is some locally detectable parameter which is approximately constant for nearby patches. So, given two modules A and B that receive input from neighboring patches, we want their outputs y_a and y_b to be approximately equal. We will describe two different noise models for this situation, which lead to slightly different, but closely related, objective functions. Both models produce equivalent behaviour in practice

[1] But if the number of weights in each module is much larger than the number of output units, then this cost is insignificant relative to the total complexity of the algorithm, which includes the cost of updating the states in the network and back-propagating the derivatives.

(and in fact, as we shall see in the next section, they are formally equivalent), but the second is conceptually cleaner. However, since we used the first model in our preliminary experiments, we will present both models below.

4.1.1 Model I.

If we think of y_b as a signal that we are trying to predict, and y_a as a version of that signal that is corrupted by additive, independent, Gaussian noise, and we assume that both y_a and y_b have Gaussian distributions, then the information that y_a provides about y_b is determined by the log ratio of two variances (Shannon, 1948) :

$$I_{y_a;y_b} = 0.5 \log \frac{V(sig + noise)}{V(noise)} \tag{4.2}$$

In this model, where y_a is a noisy estimator of y_b, the information rate is:

$$I_{y_a;y_b} = \log \frac{V(y_a)}{V(y_a - y_b)} \tag{4.3}$$

So, for y_a to provide a lot of information about y_b we need y_a to have high variance but $y_a - y_b$ to have low variance. For symmetry, we actually optimize the following objective function (where κ is a small constant which prevents the information measure from becoming infinitely large):

$$I_{y_a;y_b} \tilde{=} 0.5 \left(\log \frac{V(y_a)}{V(y_a - y_b) + \kappa} + \log \frac{V(y_b)}{V(y_a - y_b) + \kappa} \right) \tag{4.4}$$

In the above model, modules A and B are actually making inconsistent assumptions about each other's output distributions (Ralph Linsker, *personal communication*). Theoretically, $I_{y_a;y_b}$ should be equal to $I_{y_b;y_a}$. However, each module is assuming that it is conveying a noisy version of some signal, and that the other is conveying the pure signal. Under these assumptions, A would predict its own output to have higher variance than y_b, while B would predict its own output to have higher variance than y_a. Thus, the assumptions cannot both be valid except in the limit as the variance of $y_a - y_b$ approaches zero. However, in practice this is not a problem, because each module is minimizing $V(y_a - y_b)$, so our model becomes more accurate as learning proceeds.

4.1.2 Model II.

A better model would start with the assumption that both y_a and y_b are noisy versions of the same underlying signal, each with independent additive Gaussian noise:

$$y_a = sig + noise_a$$
$$y_b = sig + noise_b$$

This alternative model has the advantage that modules A and B are now making consistent assumptions about each other. The mutual information between two Gaussians, y_a and y_b, can be computed as follows, using the definitions of the entropies of Gaussians from Chapter 1:

$$\begin{aligned} I_{y_a;y_b} &= H(y_a) + H(y_b) - H(y_a, y_b) \\ &= \log \sqrt{2\pi\, e\, V(y_a)} \; + \; \log \sqrt{2\pi\, e\, V(y_b)} - \log \left(2\pi\, e\, \left|\mathbf{Q}_{\mathbf{y},\mathbf{y}}\right|^{1/2} \right) \\ &= 0.5 \log \frac{V(y_a)V(y_b)}{\left|\mathbf{Q}_{\mathbf{y},\mathbf{y}}\right|} \end{aligned} \tag{4.5}$$

where $\mathbf{y}$ is the column vector $[y_a, y_b]$, and $\left|\mathbf{Q}_{\mathbf{y},\mathbf{y}}\right|$ is the determinant of its covariance matrix. Under our model, assuming the noise terms $noise_a$ and $noise_b$ have means of zero, this determinant can be computed as follows:

$$\begin{aligned} \left|\mathbf{Q}_{\mathbf{y},\mathbf{y}}\right| &= \left| \left\langle (\mathbf{y} - \overline{\mathbf{y}})\, (\mathbf{y} - \overline{\mathbf{y}})^T \right\rangle \right| \\ &= \left\langle (y_a - \overline{y_a})^2 \right\rangle \left\langle (y_b - \overline{y_b})^2 \right\rangle - \langle (y_a - \overline{y_a})\, (y_b - \overline{y_b}) \rangle^2 \\ &= V(y_a)V(y_b) - \left\langle (sig + noise_a - \overline{sig})(sig + noise_b - \overline{sig}) \right\rangle \\ &= V(y_a)V(y_b) - \left\langle sig^2 + sig\ noise_a + sig\ noise_b + noise_a\ noise_b - 2\ sig\ \overline{sig} \right. \\ &\qquad\qquad \left. - noise_a\ \overline{sig} - noise_b\ \overline{sig} + \overline{sig}^2 \right\rangle \\ &= V(y_a)V(y_b) - \left\langle sig^2 - \overline{sig}^2 \right\rangle \\ &= V(y_a)V(y_b) - V(sig) \end{aligned} \tag{4.6}$$

Substituting 4.6 into 4.7, the mutual information between y_a and y_b under our model is:

$$\begin{aligned} I_{y_a;y_b} &= 0.5 \log \frac{V(y_a)V(y_b)}{V(y_a)V(y_b) - V(sig)} \\ &= 0.5 \log \frac{V(y_a)V(y_b)}{V(y_a)V(y_b) - V(\frac{y_a+y_b}{2}) + V(\frac{y_a-y_b}{2})} \end{aligned} \tag{4.7}$$

Under the same model, a simpler measure suggested by Allan Jepson (*personal communication*), which works equally well in practice, is the following:

$$\begin{aligned} I^* = I\left(\frac{y_a + y_b}{2}; sig\right) &= 0.5 \log \frac{V(sig + \frac{noise_a + noise_b}{2})}{V(\frac{noise_a + noise_b}{2})} \\ &= 0.5 \log \frac{V(sig + \frac{noise_a + noise_b}{2})}{V(\frac{noise_a - noise_b}{2})} \\ &= 0.5 \log \frac{V(\frac{y_a+y_b}{2})}{V(\frac{y_a-y_b}{2})} \\ &= 0.5 \log \frac{V(y_a + y_b)}{V(y_a - y_b)} \end{aligned} \tag{4.8}$$

This measure tells how much information the average of y_a and y_b conveys about the common underlying signal.

In our preliminary simulations, described in Sections 4.3 and 4.4 of this chapter, we used Model I, described in the previous subsection, with the objective function given by equation 4.4; this information measure gives good results in practice. In all other simulations described in this chapter, assuming the model described in this subsection, we used I^* (equation 4.8). The latter measure is somewhat simpler computationally as it involves fewer variance terms, and therefore permits faster simulations.

The derivation for the gradients of the information measures in equations 4.4 and 4.8 are given in Appendix B. The final expression we use in our learning algorithm for the partial derivative of $I_{y_i;y_j}$ (equation 4.4), the mutual information between the outputs of the ith and jth units with respect to the output of the ith unit on case α, y_i^α, is:

$$\frac{\partial I_{y_i;y_j}}{\partial y_i^\alpha} = \frac{2}{N}\left[\frac{y_i^\alpha - \langle y_i\rangle}{V(y_i)} - 2\frac{(y_i^\alpha - y_j^\alpha) - \langle y_i - y_j\rangle}{V(y_i - y_j) + \kappa}\right] \tag{4.9}$$

where $\langle y_i\rangle$ is the output of the ith unit averaged over the ensemble of N training cases.

4.2 A related statistical method

A standard statistical method called canonical correlation (Mardia, Kent and Bibby, 1979) is closely related to the continuous Imax objective functions we have presented (Equations 4.4, 4.7, and 4.8). We show in this section that in the linear case, all of these methods are equivalent. We then present simulation results comparing canonical correlation to continuous Imax on easy (i.e. linearly separable) and hard versions of the binary shift problem.

Given two sets of variables, $\mathbf{x_a}$, and $\mathbf{x_b}$ (not necessarily having the same dimensionality), the object of canonical correlation analysis is to find linear combinations $y_a = \mathbf{w_a^T x_a}$ and $y_b = \mathbf{w_b^T x_b}$ that will maximize the correlation between y_a and y_b:

$$\begin{aligned}\rho(y_a, y_b) &= \frac{\langle (y_a - \overline{y_a})(y_b - \overline{y_b})\rangle}{\left(\langle (y_a - \overline{y_a})^2\rangle \langle (y_b - \overline{y_b})^2\rangle\right)^{1/2}} \\ &= \frac{\mathbf{w_a}^T \mathbf{Q}_{\mathbf{x_a},\mathbf{x_b}} \mathbf{w_b}}{\left(\mathbf{w_a}^T \mathbf{Q}_{\mathbf{x_a},\mathbf{x_a}} \mathbf{w_a} \mathbf{w_b}^T \mathbf{Q}_{\mathbf{x_b},\mathbf{x_b}} \mathbf{w_b}\right)^{1/2}}\end{aligned} \tag{4.10}$$

where $\mathbf{Q}_{\mathbf{x_a},\mathbf{x_a}} = \left\langle (\mathbf{x_a} - \overline{\mathbf{x_a}})(\mathbf{x_a} - \overline{\mathbf{x_a}})^T \right\rangle$, $\mathbf{Q}_{\mathbf{x_b},\mathbf{x_b}}$ is computed similarly, and $\mathbf{Q}_{\mathbf{x_a},\mathbf{x_b}} = \left\langle (\mathbf{x_a} - \overline{\mathbf{x_a}})(\mathbf{x_b} - \overline{\mathbf{x_b}})^T \right\rangle$. Since ρ does not depend on the scale of y_a and y_b, we can equivalently maximize $\langle (y_a - \overline{y_a})(y_b - \overline{y_b})\rangle = \mathbf{w_a}^T \mathbf{Q}_{\mathbf{x_a},\mathbf{x_b}} \mathbf{w_b}$ subject to $V(y_a) = V(y_b) = 1$. Further, since ρ is invariant with respect to translations of its arguments, we can constrain the variables y_a and y_b to have zero means as well as unit variances, and simply maximize $\langle y_a y_b\rangle$.

The Imax objective functions are also invariant with respect to scaling and translations of the variables y_a and y_b. Therefore, we can constrain the variables to have means of zero and unit variances. If we do so, we find that:

$$\begin{aligned}V(y_a + y_b) &= \left\langle (y_a + y_b)^2 \right\rangle \\ &= \left\langle y_a^2 + y_b^2 + 2 y_a y_b \right\rangle \\ &= 2\left(1 + \langle y_a y_b\rangle\right) \\ V(y_a - y_b) &= \left\langle (y_a - y_b)^2 \right\rangle \\ &= \left\langle y_a^2 + y_b^2 - 2 y_a y_b \right\rangle\end{aligned} \tag{4.11}$$

$$= \quad 2\left(1 - \langle y_a y_b \rangle\right) \tag{4.12}$$

From this, it is easy to show that maximizing any of the continuous Imax objective functions (Equations 4.4, 4.7, and 4.8) is equivalent to maximizing $\langle y_a y_b \rangle$ subject to $V(y_a) = V(y_b) = 1$ and $\overline{y_a} = \overline{y_b} = 0$. Therefore, all the continuous Imax objective functions presented in this chapter are equivalent to each other, and in the linear case, our continuous Imax algorithm is equivalent to canonical correlation. In the general case, however, Imax is not equivalent to canonical correlation, since in the Imax algorithm, y_a and y_a are computed by nonlinearly transforming the input variables $\mathbf{x_a}$ and $\mathbf{x_b}$. As we shall see in the following subsection, this nonlinearity is crucial in solving certain problems.

4.2.1 Canonical correlation compared to Imax on the shift problem

The optimal solution to the canonical correlation problem can be obtained analytically as follows (Mardia, Kent and Bibby, 1979) ; first, we define the following matrices:

$$\begin{aligned}
\mathbf{K} &= \mathbf{Q}_{\mathbf{x_a},\mathbf{x_a}}^{-1/2} \mathbf{Q}_{\mathbf{x_a},\mathbf{x_b}} \mathbf{Q}_{\mathbf{x_b},\mathbf{x_b}}^{-1/2} \\
&= (\alpha_{\mathbf{1}}, \ldots, \alpha_{\mathbf{k}})\, \mathbf{D}\, (\beta_{\mathbf{1}}, \ldots, \beta_{\mathbf{k}})^T && (4.13) \\
\mathbf{N_1} &= \mathbf{K}\mathbf{K}^{\mathbf{T}} && (4.14) \\
\mathbf{N_2} &= \mathbf{K}^{\mathbf{T}}\mathbf{K} && (4.15)
\end{aligned}$$

where $\alpha_{\mathbf{i}}$ and $\beta_{\mathbf{i}}$ are the standardized eigenvectors of $\mathbf{N_1}$ and $\mathbf{N_2}$ respectively, or equivalently, the left and right eigenvectors of the singular value decomposition (SVD) of $\mathbf{K}$, and $\mathbf{D}$ is the diagonal matrix of the square roots of the corresponding eigenvalues. Note that $\mathbf{Q}_{\mathbf{x_a},\mathbf{x_a}}^{-1/2}$ can be computed straightforwardly from the eigen decomposition of $\mathbf{Q}_{\mathbf{x_a},\mathbf{x_a}}$.[2]

The canonical correlation vectors for $\mathbf{x_a}$ and $\mathbf{x_b}$ are:

$$\mathbf{w_{ai}} = \mathbf{Q}_{\mathbf{x_a},\mathbf{x_a}}^{-1/2} \alpha_{\mathbf{i}} \tag{4.16}$$

$$\mathbf{w_{bi}} = \mathbf{Q}_{\mathbf{x_b},\mathbf{x_b}}^{-1/2} \beta_{\mathbf{i}} \tag{4.17}$$

The canonical correlation variables with maximal correlation are formed by transforming the original variables $\mathbf{x_a}$ and $\mathbf{x_b}$ using the canonical correlation vectors $\mathbf{w_{aj}}$ and $\mathbf{w_{bj}}$ corresponding to the maximal eigenvalue $\mathbf{D}_{jj}$ in the SVD of $\mathbf{K}$:

$$y_a = \mathbf{w_{aj}}^T \mathbf{x_a} \tag{4.18}$$

$$y_b = \mathbf{w_{bj}}^T \mathbf{x_b} \tag{4.19}$$

We compared the performance of canonical correlation and Imax (using the objective function given in equation 4.8) on two instances of the binary shift problem. The first was linearly separable, and the second was not. (See Chapter 3 for a discussion of linear separability, and why the shift problem is hard in the

[2] In general, if $\mathbf{A} = \mathbf{\Gamma}\mathbf{\Lambda}\mathbf{\Gamma}^{\mathbf{T}}$, where A is a symmetric matrix, $\mathbf{\Gamma}$ is an orthogonal matrix whose columns are the standardized eigenvectors of $\mathbf{A}$, and $\mathbf{\Lambda}$ is a diagonal matrix of corresponding eigenvalues of $\mathbf{A}$, then $\mathbf{A}^{-\mathbf{1/2}} = \mathbf{\Gamma}\mathbf{\Lambda}^{-1/2}\mathbf{\Gamma}^T$ (Mardia, Kent and Bibby, 1979).

general case.)

We create each binary shift pattern by concatenating an m-bit binary vector with a shifted version of that vector. We start with a set of 12 four-bit subpatterns, created by removing from the full set of 16 possible vectors the four which are ambiguous when shifted (0000, 1111, 0101, and 1010). We then create a set of 16-bit patterns consisting of two 8-bit receptive fields; each 8-bit receptive field consists of one of the four bit subpatterns in the left half, and a shifted version in the right half. The total 16-bit patterns are created by taking the cross-product of the set of all possible left-shifted patterns with itself, combined with the cross-product of the set of all possible right-shifted patterns with itself.

To create an easy, linearly separable instance of the shift problem, we use shift *without wrap-around*, filling in the end-bits with zeroes. An example of one of these patterns (right-shifted) is:

1100 0011
0110 0001

To create a hard version of the shift problem, we include all patterns from the easy set described above, plus the set of patterns created by shifting and filling in the end-bits with ones instead of zeros. This pattern set is a superset of the set of patterns shifted *with wrap-around*, so it is as hard as the wrap-around shift-problem.

When we apply canonical correlation analysis to the easy class of input patterns described above, we find that the canonical correlation vectors for the two receptive fields are:

$$\begin{aligned}\mathbf{w_a} = \mathbf{w_b} &= [-0.231455,\ 0.694363,\ -0.694366,\ 0.231455,\\ &\quad -0.801782,\ 0.462911,\ -0.462909,\ 0.801785]\end{aligned}$$

and the correlation between each of the resulting canonical correlation variables and the shift is $\rho(y_a, shift) = \rho(y_b, shift) = 0.92$. We also applied the Imax algorithm, using a single layer network of two linear units, to the same problem to verify that Imax finds the same solution iteratively. The correlations between the outputs of the network and the shift, on ten runs of Imax, were very close to the canonical correlation solution, as shown in Table 4.1.[3]

When we apply canonical correlation analysis to the hard class of input patterns described above, we find that the canonical correlation vectors (obtained analytically, as described above) for the two receptive fields are all zero. Thus, there is no possible linear transformation of the input vectors which will result in correlated variables in this case. On the other hand, when we apply Imax, using a multi-layer network (consisting of two modules, each with 10 nonlinear hidden units and one linear output unit) to this problem, on ten runs, we get outputs which are perfectly correlated with the shift, as shown in Table 4.2. On each run, the hidden layer of the network was bootstrapped (as described in Chapter 3) for ten steepest descent iterations, and then the entire network was trained for ten conjugate gradient iterations.

[3] The Imax correlations are in fact slightly higher (in the third decimal place) than that achieved by canonical correlation. This can probably be attributed to the loss of numercial precision involved in the computation of the canonical correlation variables (i.e., the computation of the inverse square roots of the correlation matrices, their eigenvalues, etc. as described above).

$\rho(y_a, shift)$
-0.92582
0.925819
0.92582
-0.92582
0.925822
-0.92582
0.92582
-0.925818
0.925821
0.925821

Table 4.1: The correlation between one of the output unit's activities and the shift, after learning, on 10 runs, when a single-layer linear network was trained with Imax on the easy shift problem.

$\rho(y_a, shift)$
-0.999921
0.993861
0.999608
-0.998067
0.998734
0.998166
-0.99953
-0.998192
0.999416
0.998595

Table 4.2: The correlation between one of the outputs of a multi-layer network (with nonlinear hidden units) and the shift, after learning, on 10 runs, when trained with Imax on the hard binary shift problem.

4.3 Discovering real-valued depth for fronto-parallel surfaces

A good problem domain for testing our algorithm is that of learning to represent relative depth (stereo disparity) in stereo images with continuously varying, real-valued depths. As in the discrete case (discussed in the previous chapter), this problem is not linearly separable; a network of sigmoid units (defined in equation 1.3) requires hidden units to extract disparity unambiguously.

In stereo images of natural scenes, there are many spatially coherent features in addition to depth which the algorithm could learn, such as surface orientation, texture, colour, and brightness. In order to restrict the problem to pure stereo disparity, we use random dot stereograms. These are created by completely covering a three-dimensional scene/surface with a random dot texture, so that all information about the scene besides the depth is removed; the scene is then projected onto two different stereo views, so that each view alone is completely random. Taken together, the stereo images can be used to reconstruct the original (relative) depths of the scene, by computing the disparity between corresponding dots in the two views.

In the following simulations, we use random dot stereograms of curved surfaces, as shown in figure 4.1, and modules with deterministic, real-valued outputs that learn to represent real-valued depths (disparities) with sub-pixel accuracy. A slice of a curved surface is generated by randomly choosing a collection of control points (scaled to lie within a depth range of $[-1, 1]$), and fitting a curve through these points using cubic spline interpolation (using the "Numerical Recipes" software package, Press et al, 1988), as shown in figure 4.1. The curve can be thought of as the depth (perpendicular to the image) of a gently curved surface as a function of the (horizontal) x-coordinate in the image, for a fixed y-coordinate. Points are pseudo-randomly scattered on the curve by dividing the x-axis into a fixed number of intervals, and randomly placing a point within each interval, to create a pattern of random features.

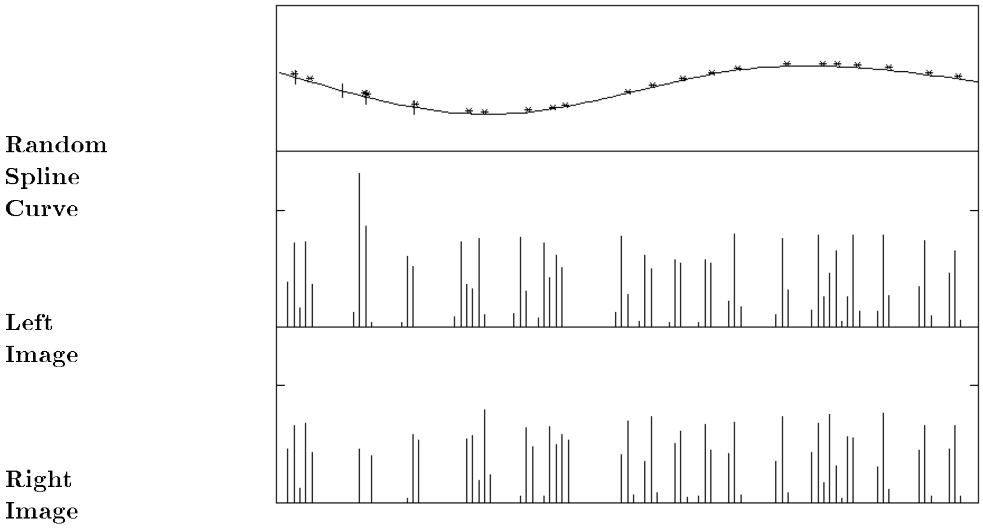

Figure 4.1: **Top:** *Part of a cubic spline fitted through seven randomly chosen, evenly spaced control points, with randomly located features scattered on it.* **Bottom:** *The "intensity" values in the two images of the surface strip. The images are made by taking two slightly different projections of the feature points, filtering the projections through a Gaussian, and sampling the filtered projections at equally spaced sample points. The sample values in corresponding patches of the two images are used as the inputs to a module. The boundaries of two neighboring patches are shown on the spline. The depth of the surface for a particular image region is directly related to the disparity between corresponding features in the left and right patch. In all our image ensembles, disparity ranges continuously from -1 to $+1$ image pixels, except where otherwise indicated in the text.*

Stereo images are formed from this pattern by taking two projections from slightly different viewing angles.[4] Finally, we blur each point through a Gaussian filter (with standard deviation approximately equal to one image pixel), sum the filter responses, and sample these blurred patterns at equally spaced intervals to create the discretized real-valued n-dimensional input vectors. The Gaussian blurring causes a single point from the random dot pattern to spread over several pixels in the resulting image. Each final image is generated by starting with a larger initial image 200 pixels wide, and randomly choosing a 100 pixel region in order to eliminate boundary effects of the cubic splines at the image edges. The final pair of n-pixel stereo patterns is treated as a 2 by n image, which is divided into smaller stereo receptive fields. The spacing of the random features is chosen so that, on average, each receptive field will contain about two features, and never less than 1 feature. (We discuss the problem of missing or unreliable data in the final section.)

Figure 4.1 shows how we generate stereo images of curved surface strips. The same technique can be applied to generate images of fronto-parallel planar surfaces. In our preliminary experiments, we used 1000

[4] We used very simple, non-perspective projections onto the $z = x$ and $z = -x$ lines of sight.

training cases of this simpler type of input, and disparity ranged from 0 to 2 pixels. We trained a network that contained 10 modules of the sort shown in figure 4.2a). Each module tried to maximize $I_{y_a;y_b}$ (equation 4.4) between the outputs of adjacent modules y_a and y_b. Each module had a single linear output unit, 10 nonlinear hidden units, and a receptive field consisting of a 2 by 8 pixel patch (2 corresponding 8 pixels strips from the left and right images). Neighboring modules' receptive fields were separated by a gap 3 pixels wide.

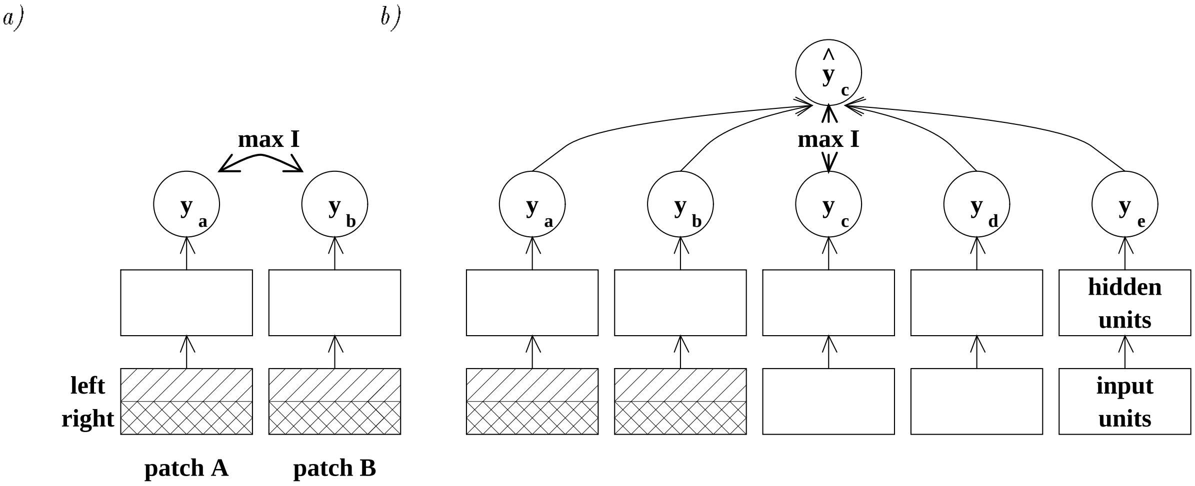

Figure 4.2: *a) Two modules that receive input from corresponding parts of stereo images. The first module receives input from stereo patch A, consisting of a horizontal strip from the left image (striped) and a corresponding strip from the right image (hatched). The second module receives input from an adjacent stereo patch B. The modules try to make their outputs, y_a and y_b, convey as much information as possible about some underlying signal (i.e., the depth) which is common to both patches. b) The architecture of the interpolating network, consisting of multiple copies of modules like those in a) plus a layer of interpolating units. The network tries to maximize the information that the locally extracted parameter y_c and the contextually predicted parameter $\hat{y}_c$ convey about some common underlying signal. We actually used 10 modules and the central 6 modules tried to maximize agreement between their outputs and contextually predicted values. We used weight averaging to constrain the interpolating function to be identical for all modules.*

Each update of the weights involves two complete passes through the training set. In the first pass, we compute the mean and variance of each output value and the mean and variance of each pairwise difference between output values given the current weights (see Appendix B). In the second pass we compute the derivatives of I (w.r.t. each unit's output) as shown in equation 4.9 for the output units of each pair of adjacent modules. We then back-propagate these terms to the hidden layer, and for each unit, we use these derivatives to accumulate dI/dw for each weight w. Then we update all the weights in parallel. After each weight update, we average corresponding weights in all the modules in order to enforce the constraint that every module computes exactly the same function of its input vector.

Without these equality constraints among groups of weights, the algorithm does not do very well at learning to extract depth (for the size of the training set we use). For small training sets, in random dot patterns, there are spurious weak correlations between pixels across the image, and between sets of pixels in neighboring patches. Each module usually learns some of the random structure that it finds in common with the adjacent neighbor. For very small training sets, this effect can be much easier to learn than effect of disparity, especially when the network starts out with random initial weights. This is because it is much easier for two units to learn to model simple correlations between pairs of their input lines than it is to learn a higher order feature such as disparity, even though the latter leads to more globally optimal solutions. Modelling the correlation between a pair of pixels can be done by a single linear unit. In contrast, disparity detection is requires hidden units, as discussed in Chapter 3. When we average weight updates among groups of units receiving input from different input patches, the effect of the low order random structure is reduced, since only the disparity is common to all input patches. (Note that we dealt with this problem differently in the binary case described in Chapter 3, by allowing each unit to maximize mutual information with *many* other units rather than just its immediate neighbors; we could do this because the disparity was constant across the image, corresponding to a single fronto-parallel planar surface, so there was long-range coherence of disparity.)

To accelerate the convergence, we used a simple line search. Each learning iteration consists of two phases: *1)* computation of the gradient, as described above, and *2)* computation of an appropriate learning rate η. At the beginning of each iteration, η is increased by a factor of 1.05. In phase 2, we first increment the weights by a proportion η of the gradient, and then if the average amount of mutual information between adjacent output units has decreased, we repeatedly backtrack, halving η each time, until the average information is greater than or equal to its previous value.

Before applying this learning algorithm, we first trained the hidden layer only, with 100 bootstrapping learning iterations, maximizing $I_{y_a;y_b}$ between pairs of hidden units in adjacent modules.[5]

We then trained the 10 modules, as described above, for 1400 iterations. Each module became highly tuned to disparity, as shown in figure 4.3a.

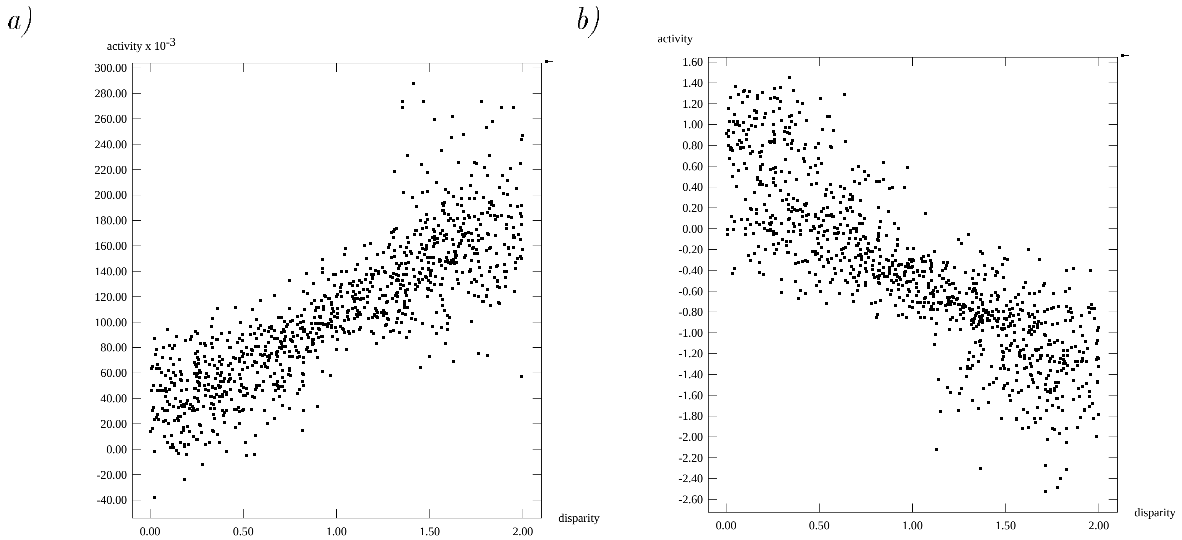

Figure 4.3: *The activity of the linear output unit of a module (vertical axis) as a function of the disparity (horizontal axis) for all 1000 test cases using planar surface strips. a) shows the response for a network trained with 10 adaptive nonlinear hidden units per module, and b) for a network trained with 100 nonadaptive Gaussian hidden units per module.*

[5] As in the binary case, we do not want two hidden units within the same module to directly or indirectly optimize their mutual information, for they could learn trivial solutions by each optimizing their mutual information with the same unit in another module. Hence we arbitrarily number the n units within the hidden layer of each module, and only allow hidden units in different modules to communicate with each other if they have been given the same number.

4.4 Ways of speeding the learning

4.4.1 Radial basis functions

We experimented with an additional method to accelerate learning, in which the first hidden layer of adaptive units in each module is replaced by a large number of non-adaptive radial basis functions (RBFs) (typically Gaussians). As discussed in Chapter 2, Moody and Darken (1989) and others have shown that a preprocessing layer of RBFs can dramatically speed subsequent supervised learning. We wished to investigate whether this preprocessing could also improve the speed of our unsupervised learning procedure.

Each radial basis unit has a "center" $\overline{\mathbf{x}}$, that is equal to a typical input vector selected at random, and gives an output y, which is a Gaussian function of the distance between the current input vector and the unit's "center":

$$y = \frac{1}{\sqrt{2\pi}\sigma} e^{-||\mathbf{x}-\overline{\mathbf{x}}||/2\sigma^2}$$

All the Gaussians had the same variance σ^2, which was chosen by hand.[6] Each module had 8 by 2 input units connected to a layer of 100 non-adaptive RBFs. Every module used exactly the same set of RBFs so that we could constrain all the modules to compute the same function.

We compared the performance of the learning with a non-adaptive hidden layer of RBFs, versus the algorithm described in the previous subsection with an adaptive layer of nonlinear hidden units, on the fronto-parallel surfaces. We found that with the RBFs, the learning took only about 50 iterations to converge (compared to 1400 for the previous network), a speedup of roughly an order of magnitude. However, there is some loss of precision in the solution; units still became highly depth-tuned, but had slightly greater variance, as shown in figure 4.3b. The difference in performance between the network with RBF hidden units and the architecture described in the next section is much more dramatic (see figure 4.5).

4.4.2 Additional equality constraints

We can improve the learning speed another way, without sacrificing precision, by further constraining the architecture as shown in figure 4.4. We can add additional equality constraints so that each module learns a set of translation-invariant feature detectors. Each module's hidden units are organized into clusters consisting of units that compute identical functions at slightly different spatial locations. Hidden units within the same cluster (shown in the same plane in figure 4.4) have corresponding weights constrained to be equal. In the following experiments, the hidden layer architecture for a module consists of four clusters of three units, so that four position-invariant feature detectors (each at 3 spatial locations) are learned by each module. [7] Additionally, as before, there are equality constraints between corresponding units in *different* modules, so that each module computes exactly the same set of feature detectors.

To speed the simulations, we adapt the weights using a conjugate gradient method with a full line search (in contrast to the "simple" line search described earlier). After 10 conjugate gradient learning iterations (typically a total of about 400 function evaluations) through a training set of 1000 patterns, the network learns to become well tuned to shift. With this architecture, bootstrapping the hidden layer (as described

[6] The variance was 0.1. The input vectors had peak values of about 0.3.

[7] The choice of four clusters of three units is somewhat arbitrary. Networks with three clusters usually are able to learn to encode disparity. Adding more clusters improves the performance, but increases the learning time.

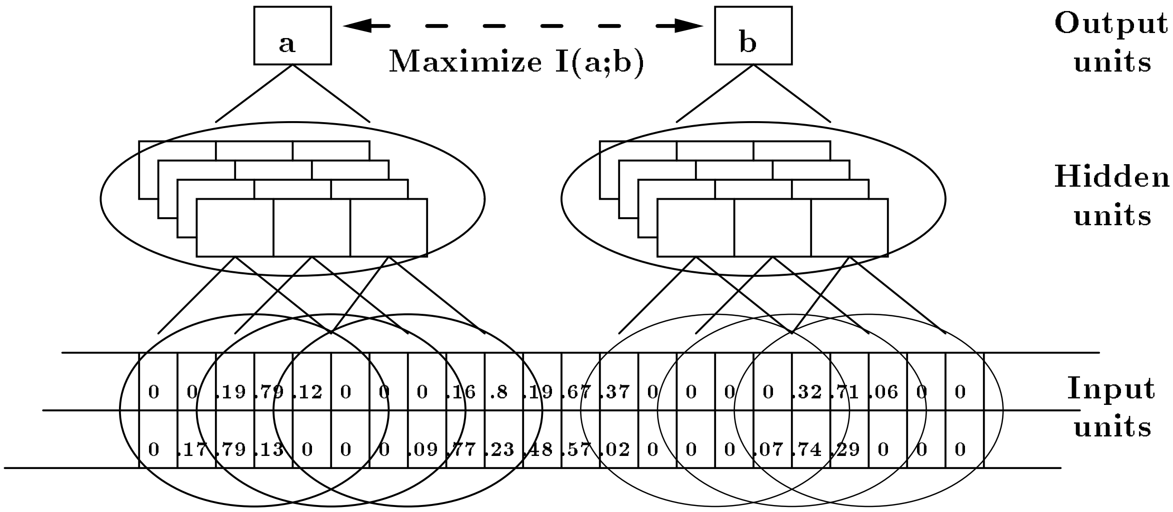

Figure 4.4: *Two modules that receive input from adjacent, non-overlapping parts of the image. Units' receptive fields are shown by ellipses. Hidden units for each module are divided into clusters. Units in the same cluster are shown in the same plane; each has a 2 by 6 pixel receptive field, and the cluster's receptive fields are offset by 2 pixels.*

in the previous chapter) is not necessary. The extra equality constraints are sufficient to force the network to find more globally coherent features, so suboptimal solutions do not seem to be a problem. Units become very well tuned to depth, as shown in figure 4.5. Typical weights learned by the hidden units of one of these modules are shown in figure 4.6.

In the experiments described in the remainder of this chapter, we use the more constrained architecture just described (i.e., with clusters of translation-invariant feature detectors in the hidden layer), the symmetric learning objective $I^* = I_{\frac{y_a+y_b}{2};sig} = \frac{V(y_a+y_b)}{V(y_a-y_b)}$, and the conjugate gradient method for learning simulations (without bootstrapping the hidden units).

4.5 More complex types of coherence

So far, we have described a very simple model of coherence in which an underlying parameter at one location is assumed to be approximately equal to the parameter at a neighboring location. This model is fine for fronto-parallel surfaces but it is far from the best model of slanted or curved surfaces. In real scenes, surface properties such as depth and shading typically vary smoothly (except at boundaries between objects). So a better model is to assume that the parameter extracted at one region can be predicted by combining the parameters extracted at *several* neighboring regions. We can apply the the same objective function I^*, as before, by assuming that the parameter at one location is an unknown linear function of the parameters at nearby locations. The particular linear function that is appropriate can be learned by the network. The network shown in figure 4.2 b) can be used to implement this model. The unit in the top layer tries to predict the locally extracted parameter, y_c, by computing an output $\hat{y}_c = w_a y_a + w_b y_b + w_d y_d + w_e y_e$ which is a linear combination of the parameters extracted from several nearby patches. The interpolating weights, as well as all the weights in the lower layers of the network, can be adjusted until the network has learned a good approximation of the depth of curved surfaces.

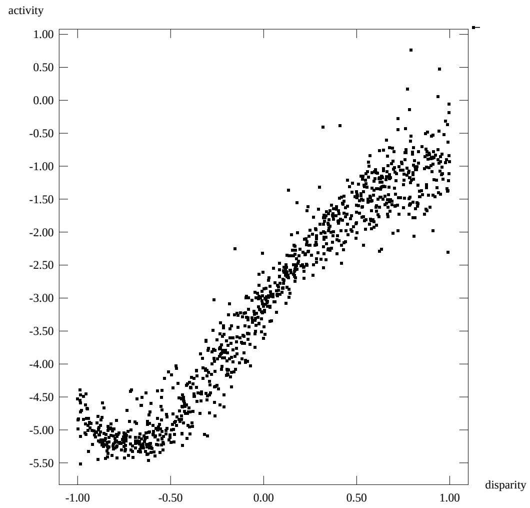

Figure 4.5: *The output of a unit in the second layer (vertical axis) as a function of the local disparity (horizontal axis) for a test set of 1000 patterns, when trained on 1000 random dot stereograms of fronto-parallel planar surface strips using the simple pairwise model, to maximize I^* with its neighbors, for 10 iterations.*

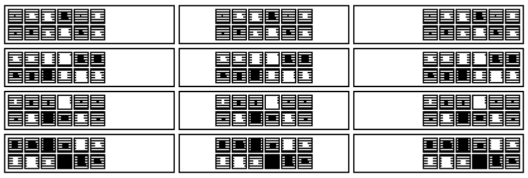

Figure 4.6: *The weights learned by the hidden units of one module on a typical run, when trained on 1000 random dot stereograms of fronto-parallel planar surface strips. Each row corresponds to the weights of units within the same cluster; these units have identical weights, and receptive fields offset by 2 pixels.*

We trained a network with ten modules of the same architecture as before (shown in figure 4.4), and an additional top layer of interpolating units as shown in figure 4.2 b)) (one interpolating unit per module, except the two modules at either end). The pattern set consisted of 1000 stereograms of curved surface strips like the one shown in figure 4.1. We first trained the lower layers of the net as before, using the simple model. The units in the second layer became reasonably well tuned to depth within 10 conjugate gradient learning iterations. After training the lower layers using the simple pairwise model, we began to train the entire network using the more elaborate, interpolating model, for 50 conjugate gradient iterations. We used constraints to force the interpolating function to be identical for all modules.

We tested this model on several image ensembles with varying amounts of curvature, by varying the spacing of the control points used to generate the cubic spline surfaces. For each data set, we simulated 3 runs starting from different initial random weights. In each case, the activity of the output units of each module became well tuned to disparity, as shown in figure 4.7 for a typical run. The network learned virtually identical interpolating weights, regardless of the initial conditions. The four weights learned for the interpolating function on typical runs, for pattern sets with varying amounts of maximum curvature, are shown in Table 4.5. For planar surfaces (2 control points), the network learns an interpolator which simply averages the four neighboring modules' outputs to predict the centre depth. As the maximum curvature *a)* *b)*

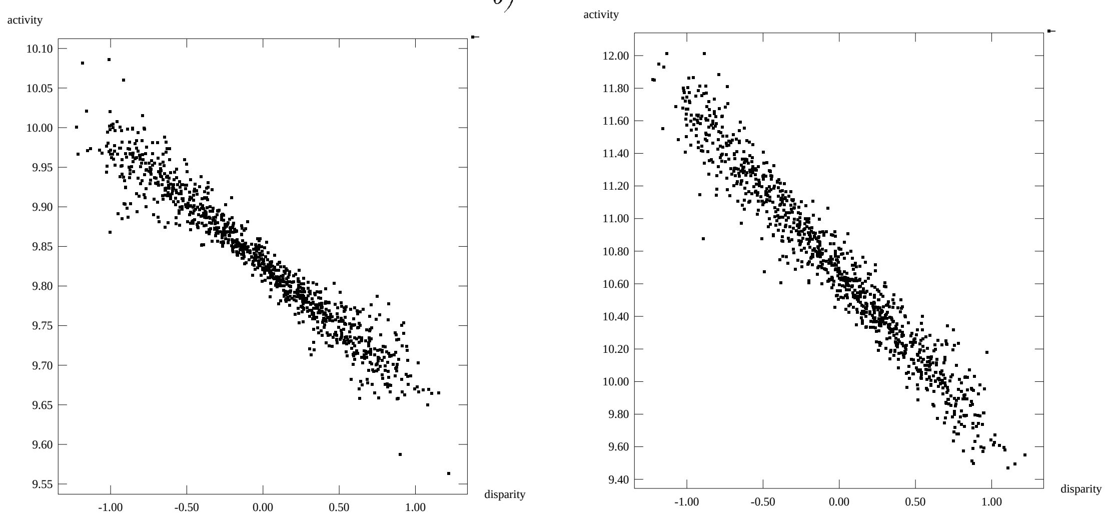

Figure 4.7: *a) The output of a unit in the second layer (vertical axis) as a function of the local disparity (horizontal axis) for a test set of 1000 patterns, when trained on 1000 curved surface strips, for 10 iterations using the simple model. b) The output of a unit in the third layer versus disparity, when trained for 50 iterations; the unit learned to predict the depth locally extracted from one module as a linear function of the outputs of the two adjacent modules on either side.*

	w_a	w_b	w_d	w_e
2 control points:	$+.256$	$+.266$	$+.258$	$+.265$
3 control points:	$+.018$	$+.516$	$+.531$	$+.011$
4 control points:	$-.147$	$+.675$	$+.656$	$-.131$
5 control points:	$-.244$	$+.734$	$+.746$	$-.256$

Table 4.3: *The four weights learned by interpolating units on typical runs, for four different pattern ensembles. The 2 control point ensemble consists of stereograms of tilted planar surfaces. For the other pattern ensembles, the number of control points shown above indicates the average number of evenly spaced control points that lie within each image, for a given image ensemble. Curvature increases with the number of control points. Each image was 118 pixels wide (comprising 10 modules' receptive fields, each 10 pixels wide, separated by 2 pixel gaps).*

increases, a characteristic pattern emerges in which positive weights are given to inputs coming from the immediately adjacent modules, and smaller negative weights are given to inputs coming from the more distant neighbors, as shown in Table 4.5. Given noise-free depth values, the optimal linear interpolator for a cubic surface is $-\frac{1}{6}, \frac{2}{3}, \frac{2}{3}, -\frac{1}{6}$, which is close to the solution learned for the 4 control point case (*exactly* a cubic surface).

4.6 Robustness of the algorithm

4.6.1 Varying the random dot density

When we increase the difficulty of the learning problem by making the random dot densities greater, the network learns a qualitatively similar interpolating function to the ones shown above, but with smaller weights. This is a sensible solution since, when predicting a depth from noisy estimates of nearby depths, the noise amplification is proportional to the sum of the squares of the weights.

Examples of patterns with varying dot densities are shown in figure 4.8. Table 4.4 shows the results of varying the random dot density for the simple model, in which we maximize I^* between the outputs of adjacent modules. The correlation between the true depth and the (standardized) output of a unit in the second layer are shown, over ten runs for each pattern set. For extremely sparse patterns in which a random dot falls within a module's receptive field about half the time, the network sometimes fails to learn to extract depth. In these cases (6 out of 10 runs), the correlation between a unit's output and the depth, as well as the mutual information I^*, are very low, and the learning seems to have become trapped in a locally optimal, very poor solution. For fairly high density dot patterns, the network still does well at extracting depth, although the correlations are not quite as high. At the highest density, the network again sometimes fails to learn to extract depth (3 out of 10 runs). The optimal dot density seems to be about 2 dots per receptive field (0.22 dots per pixel), when the receptive field width of each module is 10 pixels.

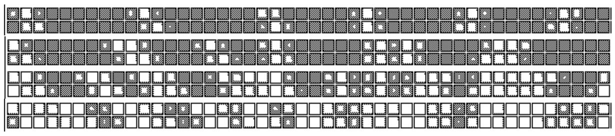

Figure 4.8: *Examples of segments of random dot stereo patterns from four different pattern ensembles, having varying dot densities. In these patterns, the probability of each pixel containing a dot (before Gaussian blurring) is (from top to bottom) .11 (1 dot randomly placed every 8-10 pixels), .22 (1 dot every 4-5 pixels; this is the density used in the previous experiments), .44 (2 dots every 4-5 pixels), and .89 (4 dots every 4-5 pixels).*

$d = .11$	$d = .22$	$d = .44$	$d = .89$
-0.021	-0.959	-0.900	-0.684
0.834	0.963	0.850	0.034
-0.038	-0.928	-0.900	-0.929
-0.579	-0.956	0.850	0.916
-0.795	-0.949	-0.935	-0.005
0.827	0.913	-0.935	-0.854
-0.039	-0.902	0.945	-0.004
0.041	-0.959	-0.920	-0.855
0.039	0.963	0.933	0.879
0.021	-0.928	0.892	-0.878

Table 4.4: *The correlation between the standardized output of a module and the shift when the network is trained on four different pattern sets, for 10 runs each, starting from different initial weights. The pattern sets had dot densities of .11, .22, .44 and .89 dots per pixel. The network was trained using the simple model, to maximize I^* between the outputs of neighboring modules.*

4.6.2 Varying the range of disparities

The curved surface pattern ensembles used in the simulations up to this point have disparities ranging from -1 to 1 pixels. As we increase the range of disparities in the pattern sets, the learning becomes much harder, especially with higher dot densities. Table 4.5 shows the performance of the simple two-layer model for pattern sets with varying ranges of disparities, when the dot density is .44 dots per pixel. With disparity ranging from -1 to 1 pixel, the network learns to become well tuned to shift on all ten runs. With larger ranges of disparities, convergence to a good solution depends much more on the initial conditions. With disparity ranging from -3 to 3 pixels, the network becomes significantly shift-tuned on only five of ten runs.

$[-1,1]$	$[-2,2]$	$[-3,3]$
-0.900	-0.873	-0.014
0.850	0.900	-0.036
-0.900	-0.870	0.034
0.850	-0.639	0.789
-0.935	-0.024	0.766
-0.935	-0.026	0.038
0.945	-0.876	-0.032
-0.920	0.871	0.792
0.933	-0.908	0.817
0.892	-0.023	0.835

Table 4.5: *The correlation between the standardized output of a module and the shift when the network is trained on three different pattern sets, for 10 runs each, starting from different initial weights. The three pattern sets had disparities ranging from -1 to 1, -2 to 2 and -3 to 3 pixels. All had dot densities of .44 dots per pixel. The network was trained using the simple model, to maximize I^* between the outputs of neighboring modules.*

4.7 Discussion

In this chapter, we have shown one way to extend the Imax algorithm to the continuous case, by making Gaussian assumptions about the underlying spatially coherent features, as well as the noise. The method was shown to work well for the problem of extracting relative depth from random dot stereograms of curved surfaces. However, the learning was shown to be sensitive to the initial conditions, particularly for sparse random dot patterns, and for pattern sets with large ranges of disparity.

Our experiments with random dot patterns of varying density show that Imax is sensitive to the dot density. The network sometimes fails to learn disparity when presented with extremely sparse patterns. On these patterns, there is less information about disparity, so the network occasionally becomes stuck in highly suboptimal solutions. One way to cope with this problem is to use a much larger data set, to provide better statistics about disparity. Adding more hidden units should also help; with more hidden units, the network has a better chance of starting out with some hidden units which are partially shift-tuned. Another possibility is to use a more robust objective function, which discounts cases that provide no information. This idea is pursued further in the next chapter when we discuss the problem of depth discontinuities, and how a mixture model of coherence and incoherence can be applied to detect such cases. However, if the learning algorithm is permitted to automatically throw out cases it cannot account for, there is a danger that it will overfit a very small subset of the data and will therefore not generalize well.

The images we have used in most of our simulations contain relatively narrow ranges of disparities (from -1 to 1). Our experiments with image ensembles containing larger ranges of disparities show that, at least for the particular network architecture used here, the network has difficulty when the range of disparities is very large. Adding more hidden units, and increasing the receptive field widths within the "hidden unit clusters", should alleviate this problem. Another solution is to have a set of output units for each module that divides up the range of disparities, each unit responding optimally to a slightly different range. In the previous chapter, we showed how this idea could be implemented using the discrete version of Imax, applied to two *sets* of units maximizing mutual information. Each unit in a set represented a different range of values of the parameter encoded by the set. In the next chapter, we show how the idea can be implementing by extending the continuous version of Imax to the case of multiple competing units, which learn a population code for depth. A third solution, discussed further in the final chapter, is to use input features at a variety of spatial scales.

We have shown how to extract a single spatially coherent parameter from an image patch. It is straightforward to generalize the I^* objective function in equation 4.8 to extract multiple parameters, by treating the parameter vector as a multi-dimensional Gaussian. Shannon showed that the information content of a multi-dimensional Gaussian signal is:

$$\log (2\pi e)^{n/2} |\mathbf{Q}|^{1/2} \tag{4.20}$$

where $|\mathbf{Q}|$ is the determinant of the covariance matrix of the signal. From this, it is easy to show that the rate at which a Gaussian multi-dimensional signal can be transmitted through a channel with Gaussian noise is:

$$0.5 \log \frac{|\mathbf{Q_{s+n}}|}{|\mathbf{Q_n}|} \tag{4.21}$$

where $\mathbf{Q_{s+n}}$ is the covariance matrix of the signal plus noise, and Q_n is the covariance matrix of the noise. We can generalize I^* to the case where $\mathbf{y_a}$ and $\mathbf{y_b}$ are multi-dimensional Gaussians as follows:

$$I_{\mathbf{y_a}+\mathbf{y_b};\mathbf{s}} = 0.5 \log \frac{\left|Q_{\mathbf{y_a}+\mathbf{y_b}}\right|}{\left|Q_{\mathbf{y_a}-\mathbf{y_b}}\right|} \tag{4.22}$$

Zemel and Hinton (1991) have applied this method to the problem of learning the viewing parameters of two-dimensional objects. The network tries to extract multi-dimensional parameters which are spatially coherent. When adjacent network modules are shown adjacent parts of the same two-dimensional object, the network is able to learn a representation of the scale, orientation and location of objects. However, the learned representation is not easily interpreted in terms of the individual viewing parameters; the network generally learns a parameter vector whose components are nonlinear combinations of scale, orientation and location. One drawback to this model is the complexity of computing the determinants of the covariance matrices. The time complexity increases as the cube of the number of elements in the parameter vector. Also, in practice, this method does poorly at finding a large number of uncorrelated parameters having very different variances (Geoffrey Hinton, 1992, personal communication); this is not surprising, as the problem of computing determinants is notoriously ill-conditioned.

The objective we have proposed for the continuous version of Imax, given in equation 4.8, is closely related to a statistical method called ACE (for Alternating Conditional Expectation – described in Hastie and Tibshirani, 1990), a nonlinear generalization of regression. ACE minimizes the squared error between a single transformed predictor variable y and an additive nonlinear transformation of a set of m observed variables x_i, over an ensemble of n observations:

$$E = \sum_{i=1}^{n}(f(y_i) - \sum_{j=1}^{m}\phi(x_{ij}))^2$$

subject to the constraint that $f(y)$ has unit variance. While ACE's underlying model differs from ours (we compute nonlinear transformations of *two sets* of variables – the two input patches – and we do not use an additive model, since the input components are not in general independent), both objectives are equivalent to maximizing the correlation between two nonlinearly transformed variables. As mentioned earlier in this chapter, this is equivalent to the canonical correlation objective function in the linear case.

Hastie and Tibshirani (1990) describe some situations in which ACE exhibits anomalous behavior. The anomalies arise particularly when y and/or x are highly non-Gaussian. For example, if x and y are jointly distributed in two disjoint clusters, then the optimal solution with respect to the ACE objective function is for $f(y)$ and $\phi(x)$ to be step functions, producing constant outputs within each cluster. Thus, ACE tends to collapse clustered data into piecewise constant functions. The continuous version of Imax presented in this chapter would be expected to behave similarly, if for example, the depth of surfaces had a bimodal or multi-modal distribution. This could arise in natural images, where depths would tend to fall into two categories, near (corresponding to objects at zero depth, at the focal plane) and far (corresponding to background objects/surfaces).

Chapter 5

Mixture models of coherence

The learning algorithms presented in Chapters 3 and 4 were based on the assumption of a single type of coherence in images. We assumed there was some parameter of the image which was either constant for nearby patches, or varied smoothly across space. In natural scenes, these simple models of coherence may not always hold. There may be widely varying amounts of curvature, from smooth surfaces, to highly curved spherical or cylindrical objects. There may be coherent structure at several spatial scales; for example, a rough surface like a brick wall is highly convoluted at a fine spatial scale, while at a coarser scale it is planar. And at boundaries between objects, or between different parts of the same object, there will be discontinuities in coherence. It would be better to have multiple models of coherence, which could account for a wider range of surfaces. One way to handle multiple models is to have a mixture of distributions. In this chapter, we explore several ways of employing mixture models to account for a greater variety of situations. We extend the learning procedure described in Chapter 4 based on these models.

5.1 Learning population codes: Competitive Imax

For the continuous version of Imax derived in the previous chapter, we assumed that the underlying spatially coherent signal has a Gaussian distribution. We discussed some of the problems associated with this assumption, when the underlying distribution is highly non-Gaussian. In a typical sample of images as seen by a typical human observer, it is not at all clear that Gaussian assumptions are valid – for depth, or any other scene parameter. Objects that the viewer fixates on would tend to lie within some relatively narrow range of distances from the observer. So many depth samples would be clustered near the viewer's focal plane. But objects not under fixation, or surfaces in the background, might equally likely have almost any depth (within the limits of the viewer's range of sensitivity). Such a distribution would look more like the one shown in figure 5.1 a). In this case, a multi-modal distribution of depths might be a better model. This can be approximated by a mixture of Gaussians model (described in Chapter 1), as shown in figure 5.1 b). We could approximate any arbitrary distribution by combining enough Gaussian distributions.

In Chapter 4, under Gaussian assumptions, we treated the output of a module as a real-valued variable which encodes the entire range of possible parameter values. An alternative encoding, requiring multiple output units, is to have each output encode a different range of a parameter. This latter approach has been called a value code (Ballard, 1986). In Chapter 3, we described one way to force a group of units to learn a value code, by extending discrete Imax to n-valued discrete variables, and assigning one unit to encode each of the values. However, this algorithm has the drawback that the time complexity for computing the statistics in the mutual information derivatives grows as the square of n. Rather than assuming that the spatially coherent image parameter is a discrete n-valued variable, we can use a mixture model, assuming that the underlying signal can be approximated by a mixture of n equally weighted Gaussians with unknown means and equal variances. We assign a separate network module to learn each component of the mixture distribution. Each of the n modules receives input from the *same* image patch, and tries to extract a parameter that agrees with a module receiving input from a neighboring patch. We decide in advance on a one-to-one correspondence between modules receiving input from adjacent image patches, as shown in figure 5.2; corresponding modules should try to make their outputs agree on each case.

We would like a group of n modules with the same receptive field to learn n models of some spatially coherent image feature, such that each module conveys maximal information about the feature *on the cases when its model holds.*

More formally, we will assume that there is some spatially coherent signal s, whose probability distribution can be modelled by a mixture of n equally weighted Gaussians having equal variances:

$$p(s) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} e^{-(s-\mu_i)^2/2\sigma^2} \tag{5.1}$$

Under this mixture model, the probability that s was generated by the ith Gaussian is:[1]

$$p_i(s) = \frac{\frac{1}{\sigma} e^{-(s-\mu_i)^2/2\sigma^2}}{\sum_j \frac{1}{\sigma} e^{-(s-\mu_j)^2/2\sigma^2}} \tag{5.2}$$

We will further assume that the n modules assigned to model s for a given input patch a produce outputs $y_{ia}, i = 1 \ldots n$, equal to shifted versions of the signal plus zero-mean Gaussian noise. The shifts are due to the biases of the units. The bias of the ith unit represents the negated mean of the ith mixture component, $-\mu_i$. The means are free to adapt, along with the rest of the weights. In so interpreting the biases, we can interpret the output of each unit, y_{ia}, as the signal shifted toward the mean of the ith mixture component, plus noise:

$$y_{ia} = s - \mu_i + noise_{ia} \tag{5.3}$$

We would like the variance of the component Gaussians to be smaller than the total variance of the signal. We can enforce this by setting the variance of each component Gaussian to be some proportion T of the actual variance of the signal. The signal variance is approximately equal to $V(y_i)$, the variance of each output (less the variance of the noise). Now we can approximately compute the probability of each of the n Gaussians having generated the signal on a particular case, purely in terms of the outputs of the group of units:

$$p_i(s) \tilde{=} \frac{e^{-{y_i}^2/(2T\ V(y_i))}}{\sum_j e^{-{y_j}^2/(2T\ V(y_j))}} \tag{5.4}$$

[1] See McLachlan and Basford (1988) for a good introduction to mixture models for arbitrary as well as Gaussian component distributions.

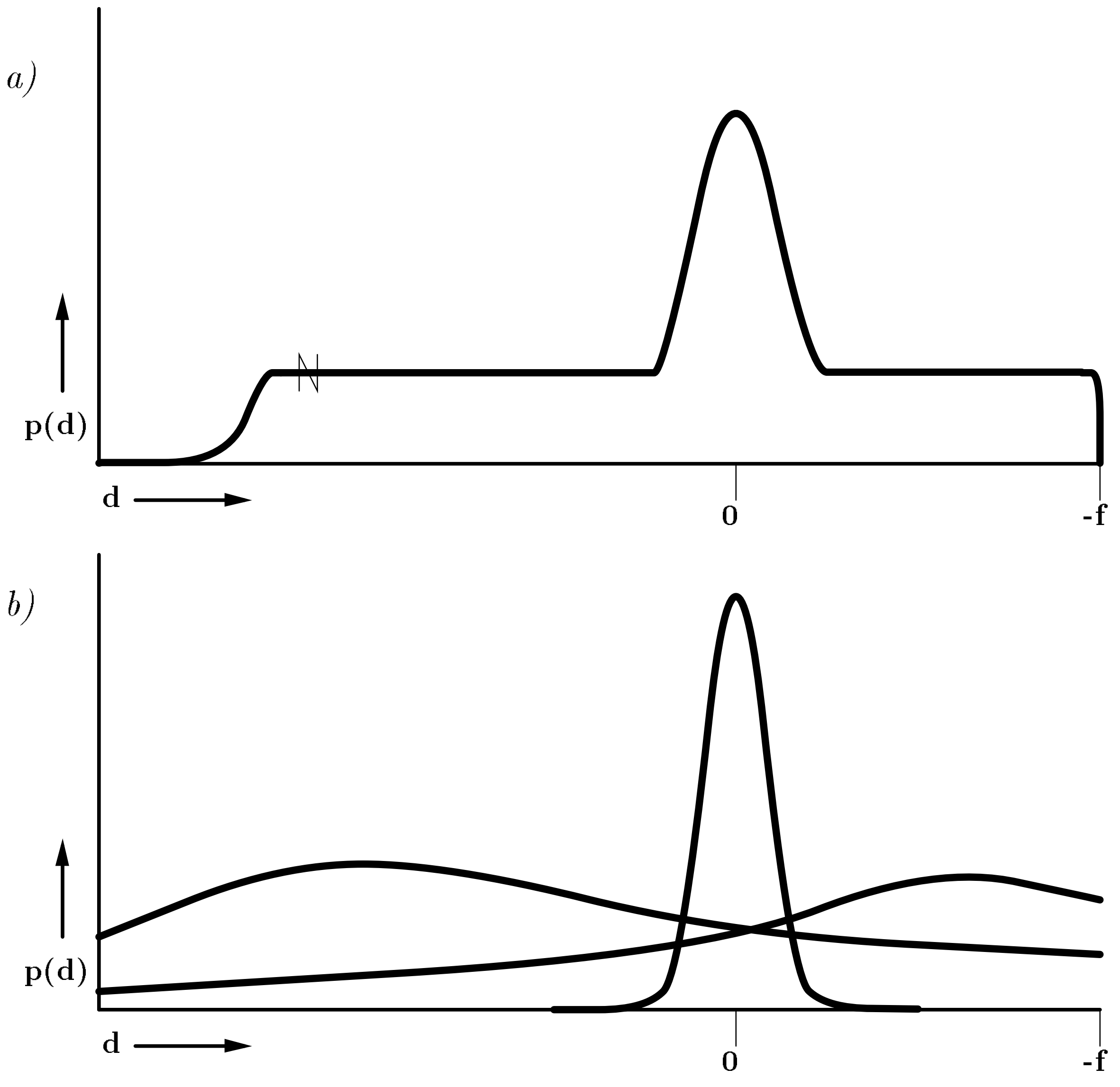

Figure 5.1: *a) A hypothesized probability distribution of depths d, p(d), in natural scenes viewed by a human observer. b) The distribution in a) is modeled as a mixture of three Gaussians, one with zero mean and small variance, and two with nonzero means and larger variances. The point $d = -f$ indicates the location of the observer (at a depth of minus the focal length, f). The point $d = 0$ indicates the focal point.*

This quantity is only an approximation because of the noise term included in y_{ia}. However, assuming equal noise variances, the approximation should be close to the true value of $p_i(s)$.

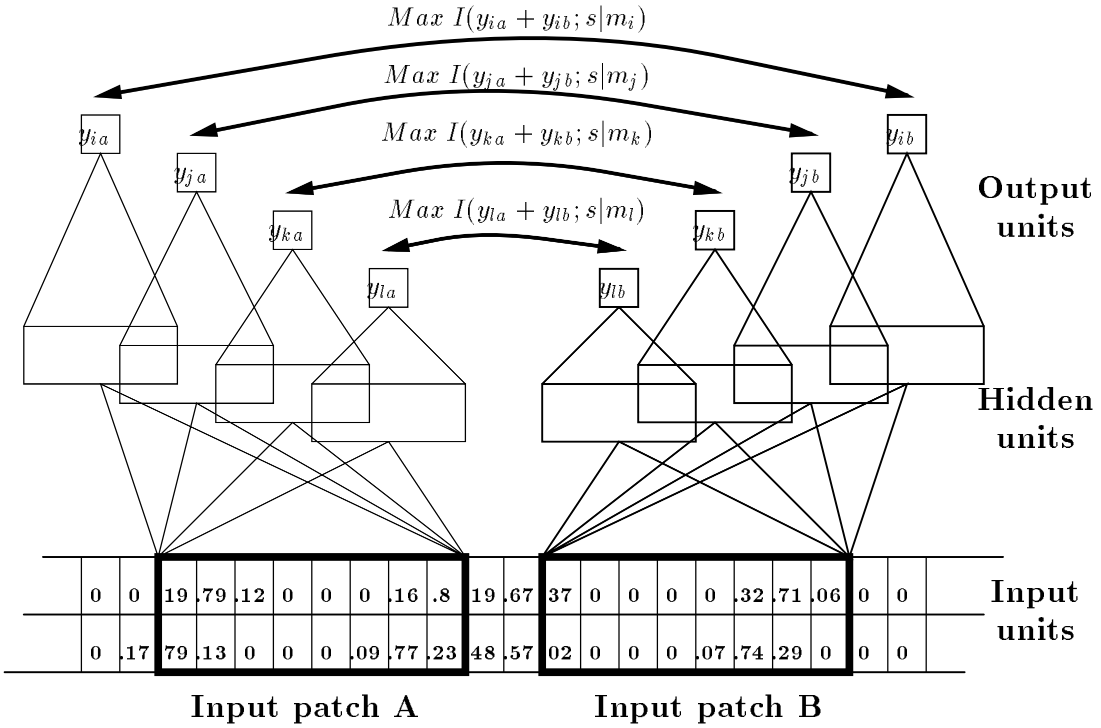

Figure 5.2: *Two groups of modules that receive input from adjacent, non-overlapping parts of a stereo image. Modules that receive input from the same image patch compete to try to model each pattern. The output units of corresponding modules that receive input from adjacent patches compute the same model; they try to maximize the information they convey about a common underlying signal when their model holds.*

Now that we have a mixture model of the signal, we need an objective function which applies to this model. The Imax objective function for continuous signals presented in Chapter 4 relied on the assumption that the signal was Gaussian. Here, the signal is not Gaussian, it is a mixture of Gaussians, so the same objective cannot be directly applied. However, if we could determine which mixture component generated the data in each case, we could select the cases corresponding to one particular model, and be assured that the signal came from one particular Gaussian distribution. We could then select the unit responsible for that model, and adapt its weights on only those cases (under Gaussian assumptions about the signal), using equation 4.8 from Chapter 4, which we repeat here:

$$I^* = I\left(\frac{y_a + y_b}{2}; sig\right) = 0.5 \log \frac{V(y_a + y_b)}{V(y_a - y_b)} \tag{5.5}$$

Although we cannot tell with complete certainty which model generated each case, we can use equation 5.4 to estimate the probability that each model holds. Given the probability that the ith model holds on each particular case, we can compute the information that the ith neighboring modules' outputs convey about the common underlying signal:

$$I_{y_{ia}+y_{ib};s \mid model\ i} = 0.5 \log \frac{V^{iab}(y_{ia} + y_{ib})}{V^{iab}(y_{ia} - y_{ib})} \tag{5.6}$$

where the V^{iab}s are the variances given that the underlying signal in patches *a and b* was generated by the ith model. These variances are computed by weighting each term by the probability that model i holds in *both* image patches, $p_{iab} = p_{ia}p_{ib}$, for that case, e.g.:

$$V^{iab}(y_{ia} - y_{ib}) = \left\langle (y_{ia} - y_{ib})^2 p_{iab} \right\rangle - \langle (y_{ia} - y_{ib})\, p_{iab} \rangle^2 \tag{5.7}$$

To prevent units from conveying infinitely high information by modelling only a small subset of the cases, each module pair's information measure is weighted by the total probability across cases of their model holding. The total objective function to maximize is:

$$I^{**} \quad = \quad \sum_{iab} \langle p_{iab} \rangle \ \log \frac{V^{iab}(y_{ia} + y_{ib})}{V^{iab}(y_{ia} - y_{ib})} \tag{5.8}$$

where $\langle p_{iab} \rangle$ is the expected value of the probability of the ith model holding in patches a and b, averaged over all cases.

So for a given pair of output units, the extent to which each training case contributes to the variances, and hence to the learning, depends on how well those two units account for the current case. This is a function of how close the signal is to the mean computed by those units. In trying to each account for as many cases as possible, the modules for a given input patch are forced to compete to form the best model of each case. The idea of having several different pairs of outputs that compete to represent the mutual information across space in the input has also been used by Zemel and Hinton (1991) for a very different task.

The derivatives of this information measure with respect to weights in the network are derived in Appendix C. To simplify the derivatives, the variances of units' outputs, $V(y_i)$, are approximated when the probabilities $p_i(s)$ are computed for equation 5.4. Assuming the activations on each output unit's incoming connections have equal variances σ^2 (which is not unreasonable, since the hidden units use the sigmoidal nonlinearity, so their outputs are between 0 and 1), the output variance of the jth unit is approximately equal to $\sum_i w_{ji}^2 \sigma^2$. This would be exactly true, given our assumptions, if the input lines were independent. So we approximate the variance of each unit's output by its summed squared weights times a constant.[2]

We tested the procedure using a network of four modules per image patch, as shown in figure 5.2, and ten image patches. Each module had six hidden units, divided into two clusters of shift-invariant feature-detectors. This basic architecture was motivated, and described in more detail in the previous chapter. As before, equality constraints were used to force corresponding modules receiving input from different patches (see figure 5.2) to compute exactly the same model. 1000 training patterns of random dot stereograms of planar surfaces were used. The patterns were created in the same manner as described in the previous

[2] The constant can be absorbed in T of equation 5.4. This approximation of the $V(y_i)$s simplifies the derivatives considerably, since $\frac{\partial p(s)}{\partial w_{ji}}$ for a given case now depends only on the group of units' outputs for that case, rather than the statistics $V(y_k)$ which must be accumulated for each output y_k across all cases.

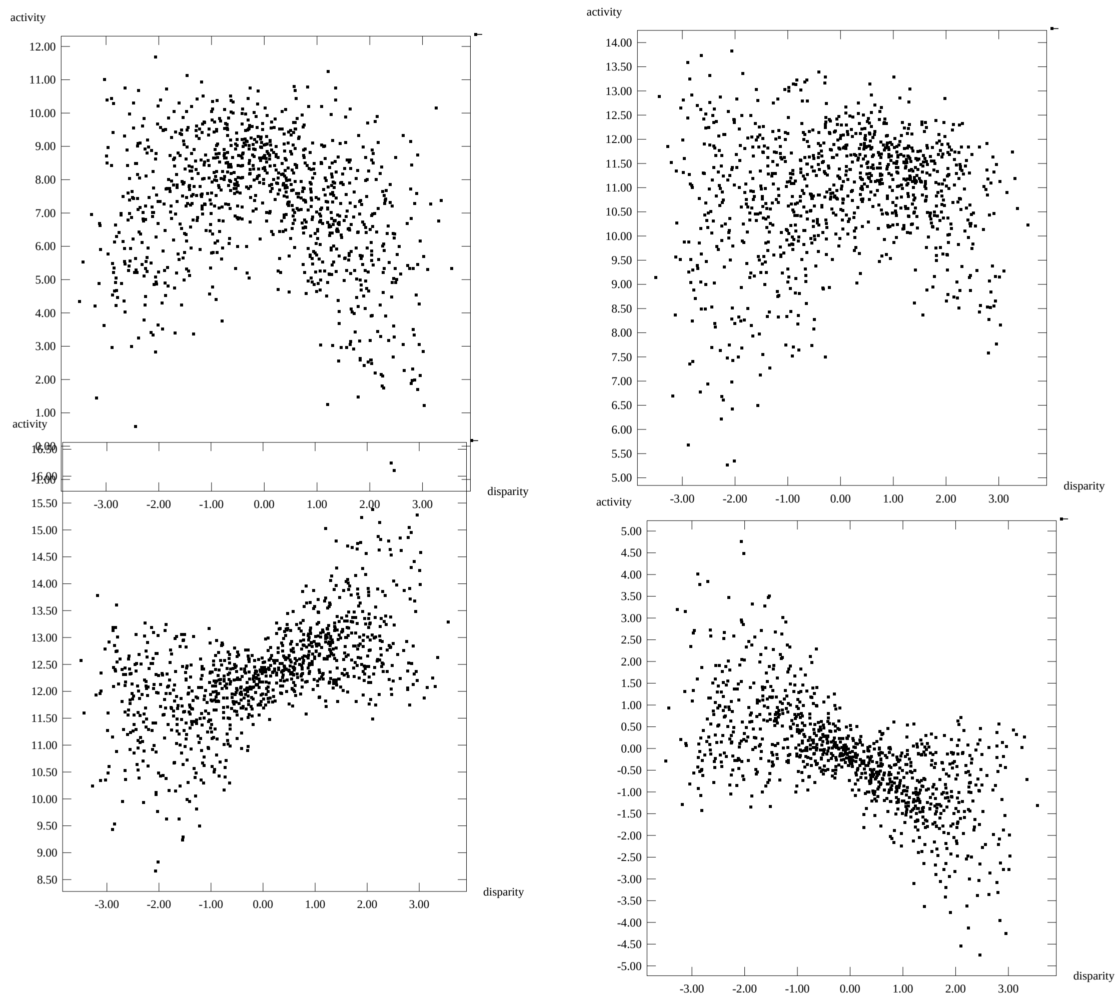

Figure 5.3: *The responses of the linear output units of four competing modules versus disparity for all 1000 test cases using planar surface strips.*

chapter, but this time, the disparity ranged continuously from -3 to $+3$ pixels. The network was trained for five conjugate gradient iterations, with $T = 100.0$.[3]

Figure 5.3 shows scatterplots of the depth-tuned outputs of four competing modules on a typical run, for a test set of 1000 patterns. Figure 5.4 shows the mean tuning curves for these scatterplots. Although the responses have high variance and range over very different scales, as shown in the scatterplots, they are all disparity tuned, and each responds best to a different range of disparities. Table 5.1 shows the correlation between the outputs of the four competing units and the shift, on five runs, starting from different initial random weights. On four out of five runs, all four output units became shift-tuned; on the third run, only two of the four output units became shift-tuned.

[3] Using a large value of T softens the competition imposed by the softmax activation function, thereby encouraging each unit to model some of the cases. Using a very small value of T usually results in only one or two of the units capturing all of the cases.

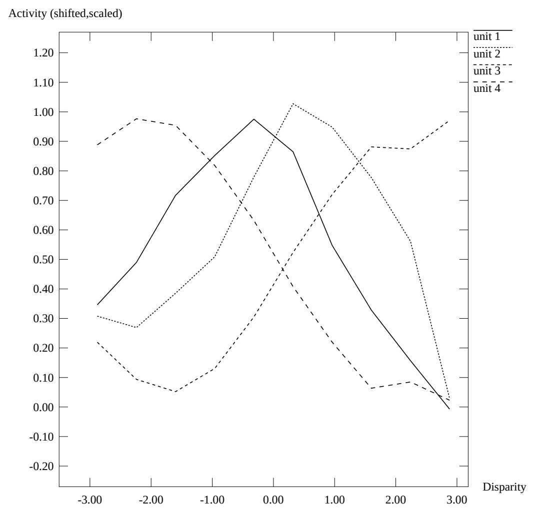

Figure 5.4: *The mean response curves for the response scattergrams shown in the previous figure. Responses were scaled and shifted to lie between 0 and 1. To compute the mean curves, the horizontal axis was divided into 10 even intervals, and average responses were computed within each interval, over a test set of 1000 patterns, for each unit.*

Unit 1	Unit 2	Unit 3	Unit 4
-0.129125	0.141367	-0.101754	0.264536
0.247533	-0.588927	0.605804	-0.529443
-0.290071	-0.0338237	-0.414734	0.00904407
0.195174	0.277245	-0.419201	-0.396053
0.190812	0.53617	0.499982	-0.295365

Table 5.1: *The correlation between the standardized outputs of four competing units and the shift is shown for five runs, each starting from different initial weights. The network was trained on fronto-parallel planar random dot stereograms, with disparities ranging from -3 to +3 pixels.*

5.2 Throwing out discontinuities

If the surface is curved, provided it is continuous, the depth at one patch can be accurately predicted from the depths of two patches on either side. We have shown how an interpolating layer can be added to a depth-extracting network, to make depth predictions over larger spatial scales, using an architecture as shown in figure 5.7. If, however, the training data contains cases in which there are depth discontinuities (see figure 5.5) the interpolator will also try to model these cases. This will not only contribute noise to the interpolating weights and to the depth estimates, but we will have no way of knowing when a discontinuity is present. One way of reducing this noise is to treat the discontinuity cases as outliers and to throw them out. Rather than making a hard decision about whether a case is an outlier, we can make a soft decision by using a mixture model of coherence. For each training case, the network compares the locally extracted depth, y_c, with the depth predicted from the nearby context, $\hat{y}_c$. The network assumes that $\hat{y}_c$ is drawn from a Gaussian having mean y_c if it is a continuity case, and from a uniform distribution if it is a discontinuity case, as shown in figure 5.6. It can then estimate the probability of a continuity case:

$$p_{cont}(\hat{y}_c) = \frac{N(\hat{y}_c, y_c, V_{cont}(\hat{y}_c))}{N(\hat{y}_c, y_c, V_{cont}(\hat{y}_c)) + k_{discont}} \tag{5.9}$$

where N is a Gaussian, and $k_{discont}$ is a constant representing a uniform density.

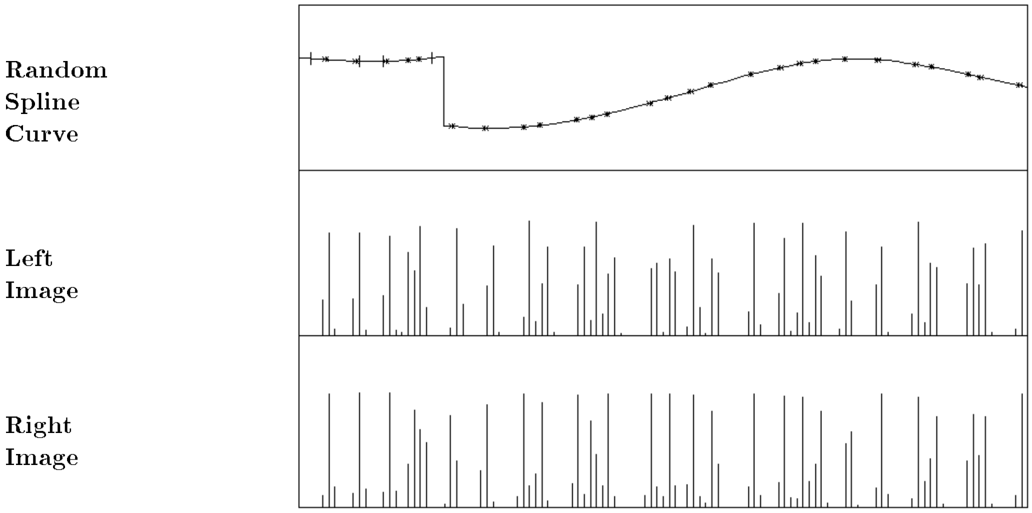

Figure 5.5: **Top:** *A curved surface strip with a discontinuity created by fitting 2 cubic splines through randomly chosen control points, 25 pixels apart, separated by a depth discontinuity. Feature points are randomly scattered on each spline with an average of* 0.22 *features per pixel.* **Bottom:** *A stereo pair of "intensity" images of the surface strip, created in the same manner as described in the previous chapter. Disparity ranges continuously from* -1 *to* $+1$ *image pixels. Each stereo image was 120 pixels wide and divided into 10 receptive fields 10 pixels wide and separated by 2 pixel gaps, as input for the network. The receptive field of an interpolating unit spanned 5 modules' receptive fields, or 58 image pixels. Discontinuities were randomly located a minimum of 40 pixels apart, so only rarely would more than one discontinuity lie within an interpolator's receptive field. Discontinuities were constrained to be large; the change in depth at a discontinuity was always at least 3/8 of the total depth range.*

We can now optimize the *average* information y_c and $\hat{y}_c$ transmit about the common underlying signal, given the continuity model. We assume that no information is transmitted in discontinuity cases, so the average information depends on the probability of continuity and on the variance of $y_c + \hat{y}_c$ and $y_c - \hat{y}_c$ measured only in continuity cases. As in the mixture model described in the previous section, the information measure is weighted by the total probability of the continuity model across cases, $P_{cont} = \langle p_{cont}(\hat{y}_c)\rangle$:

$$I_{cont} = 0.5 \;\; P_{cont} \;\; \log \frac{V_{cont}(y_c + \hat{y}_c)}{V_{cont}(y_c - \hat{y}_c)} \tag{5.10}$$

We tried several variations of this mixture approach. In the method found to work best, we use a two-phase training procedure. We first pretrain the network (including the interpolating layer) on patterns with discontinuities using the original continuous model described in the previous chapter. We then train the network using the mixture model. We empirically select a good (fixed) value of $k_{discont}$ for the mixture model, and we estimate $V_{cont}(\hat{y}_c)$ by choosing a good initial value $\hat{V}_{cont}(\hat{y}_c) = \alpha V(y_c)$ (some proportion of the initial variance of $\hat{y}_c$), and gradually shrink the variance during learning. After further adapting the weights for several iterations, the network becomes quite good at rejecting the discontinuity cases. However,

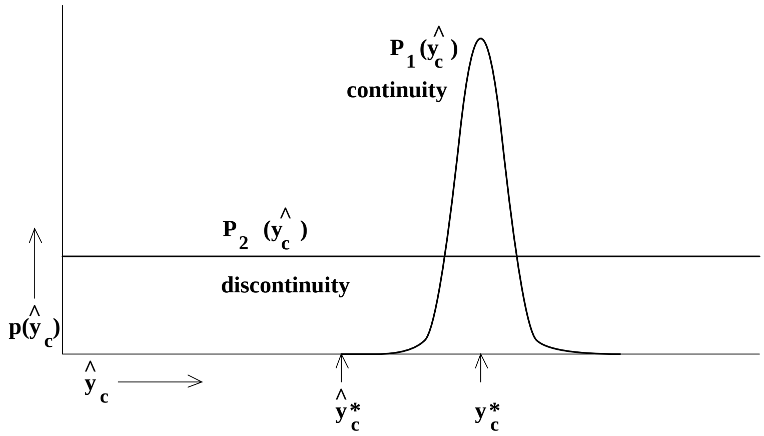

Figure 5.6: *The probability distribution of $\hat{y}_c$, $P_1(\hat{y}_c)$, is modeled as a mixture of two distributions: a Gaussian with mean $= y^*_c$ and small variance, and $P_2(\hat{y}_c)$, a uniform distribution. Sample points for $\hat{y}_c$ and y_c, $\hat{y}^*_c$ and y^*_c are shown. In this case, $\hat{y}^*_c$ and y^*_c are far apart so $\hat{y}^*_c$ is more likely to have been drawn from P_2.*

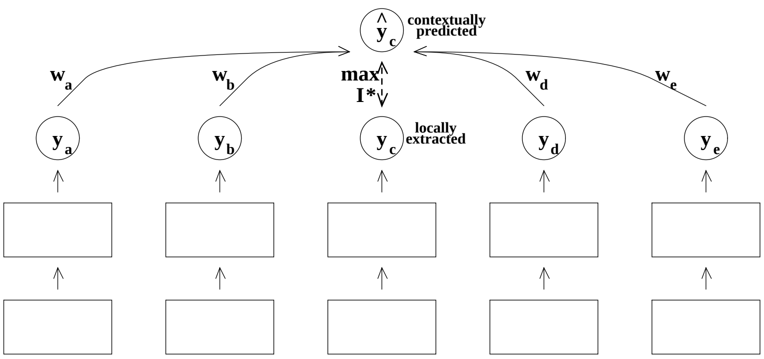

Figure 5.7: *The architecture of the interpolating network used in the previous chapter.*

this leads to only a modest improvement in the performance of the interpolator.

One drawback of this method is that the behaviour of the algorithm is dependent on good initial values for $k_{discont}$ and $\hat{V}_{cont}(\hat{y}_c)$. It is also is dependent upon good starting values for the interpolating weights (which we achieved by pretraining the network without the discontinuity model). If the values of $k_{discont}$ and $V_{cont}(\hat{y}_c)$ are allowed to adapt, or the network starts out from random initial weights (directly applying the mixture model), the network gets stuck in very poor solutions.

In cases where there is a depth discontinuity between y_a and y_b or between y_d and y_e the interpolator works moderately well because the weights on y_a and y_e are small. Because of the term P_{cont} in equation 5.10 there is pressure to include these cases as continuity cases, so they probably contribute noise to the interpolating weights. In the next section we show how to avoid making a forced choice between rejecting these cases or treating them just like all the other continuity cases.

5.3 Learning a mixture of interpolators

The presence of a depth discontinuity somewhere within a strip of five adjacent patches does not necessarily destroy the predictability of depth across these patches. It may just restrict the range over which a prediction can be made. So instead of throwing out cases that contain a discontinuity, the network could try to develop a number of different, specialized models of spatial coherence across several image patches. If, for example, there is a depth discontinuity between y_c and y_e in figure 5.7, an extrapolator with weights of $-1.0, +2.0, 0, 0$ would be an appropriate predictor of y_c.

In order to develop multiple depth interpolating models, we could apply exactly the same mixture model we used in the first section of this chapter to form population codes for depth. In this case, units might form a population of interpolators. Each interpolator would be an expert for a different class of cases. We could simply use the error in each interpolator's prediction, $\hat{y}_{ic} - y_c$, as the basis for deciding among the different models on each case. However, if we did this, the network would probably not develop several specialized interpolators for discontinuity cases. Any interpolator which applied in discontinuity cases would apply equally well in many cases with no discontinuities, particularly where the surface was close to being planar. Since the most probable model does the greatest amount of learning on each case, the network would be discouraged from developing multiple interpolators for discontinuity cases.

It would be better to have a way to detect discontinuities independent of how well the interpolators were doing. We can achieve this by adding extra "controller" units to the network, as shown in figure 5.8, whose sole purpose is to compute the probability, p_i, that each interpolator's model holds. The weights of both the controllers and the interpolating experts can be learned simultaneously, so as to maximize the same objective used in the population code model:

$$I^{**} = \sum_i \langle p_i \rangle \log \frac{V^i(\hat{y}_{ic} + y_c)}{V^i(\hat{y}_{ic} - y_c)} \tag{5.11}$$

where the V^is represent variances given that the ith model holds. By assigning a controller to each expert interpolator, each controller should learn to detect a discontinuity at a particular location (or the absence of a discontinuity in the case of the interpolator for pure continuity cases). And each interpolating unit should learn to capture the particular type of coherence that remains in the presence of a discontinuity at a particular location.

The outputs of the controllers are normalized, so that they represent a probability distribution over the interpolating experts' models. We can think of these normalized outputs as the probability with which the system selects a particular expert. Each controller's output is a normalized exponential function of its *squared* total input:

$$p_i = \frac{e^{{x_i}^2/T\ \hat{\sigma}(x_i)^2}}{\sum_j e^{{x_j}^2/T\ \hat{\sigma}(x_j)^2}} \tag{5.12}$$

Squaring the total input makes it possible for each unit to detect a depth edge at a particular location, independently of the direction of contrast change. As in the population code model, we divide the squared total input in the exponential by an estimate of its variance, $\hat{\sigma}(x_j)^2 = k \sum_{ji} {w_{ji}}^2$. This discourages any one unit from trying to model all of the cases simply by having huge weights. The controllers get to see all five local depth estimates, $y_a \dots y_e$. As before, each interpolating expert computes a linear function of four

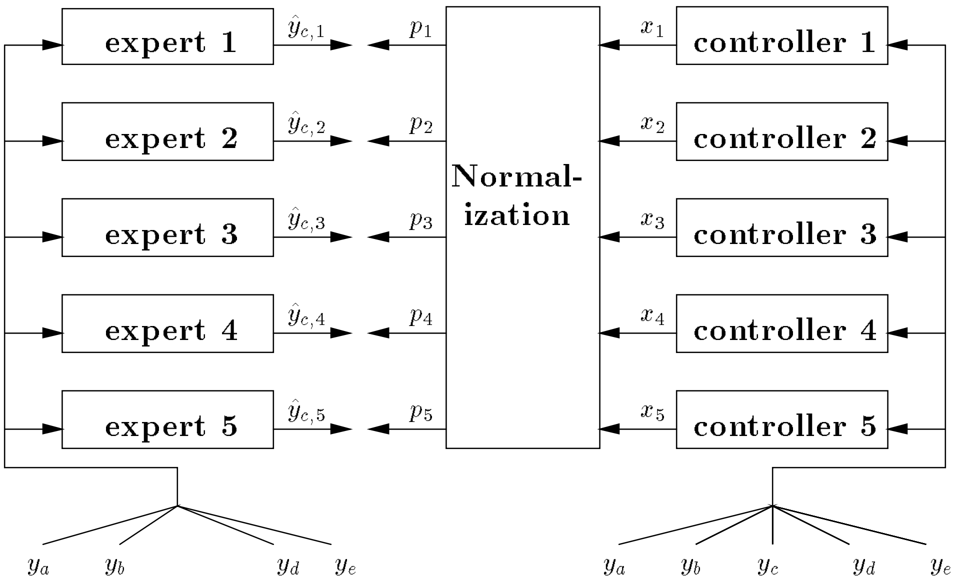

Figure 5.8: *An architecture for learning a mixture model of curved surfaces with discontinuities, consisting of a set of interpolators and discontinuity detectors. We actually used a larger modular network and equality constraints between modules, as described in Chapter 4, with 6 copies of the architecture shown here. Each copy received input from different but overlapping parts of the input.*

contextually extracted depths, $\hat{y}_{ci} = w_{ia}y_a + w_{ib}y_b + w_{id}y_d + w_{ie}y_e$, in order to try to predict the centrally extracted depth y_c.

We compared two methods of training the mixture model, the second of which turned out to be much more effective. In both methods, we first trained the network using the original continuous model (described in Chapter 4), on a training set of 1000 images with discontinuities (like the one shown in Figure 5.5). Once the lower layers of the network were tuned to depth, we then froze the weights in the first two layers, and applied the mixture of interpolators model, to maximize I^{**} for the top layer of the network.

In the first method, we only pretrained the first two layers of the network to become depth-tuned. We then froze the the weights in the lower layers, and applied the mixture model. Using the outputs of the depth extracting layer, $\ldots, y_a, \ldots, y_e, \ldots$ as the inputs to the expert interpolators and their controllers, we trained only the top level of weights in the network to maximize I^{**} (without back-propagating derivatives to the lower layers), for 10 conjugate gradient learning iterations. Figure 5.9 shows typical weights learned by the experts and by their controllers on a typical run using this method. The network learns a set of five slightly different interpolators, although the differences are barely noticeable. The bottom left unit in figure 5.9 has learned a symmetrical set of interpolating weights that applies in continuity cases. The other units learn interpolators in which the centre of gravity of the weights is shifted either to the left or right. Some of the controllers are tuned to depth edges to the left or right of the centre pixel. It appears that the depth estimates being interpolated are too noisy, so there is a trade-off between interpolating across all four inputs to improve the signal to noise ratio, and not interpolating across discontinuities.

The second training method produced better results. In this case, we applied still more pretraining to improve the depth-tuning of the layer 2 units, before applying the mixture model. Instead of just pretraining the first two layers before applying the mixture model, we pretrained all three layers, including the interpolat-

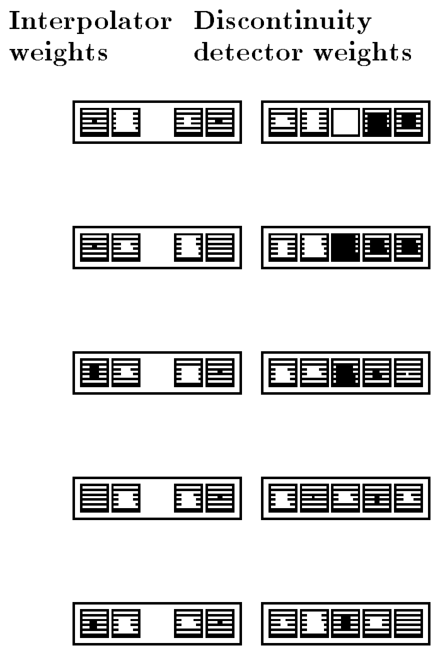

Figure 5.9: *Typical weights learned by the five competing interpolators and corresponding five discontinuity detectors, when trained to maximize I^{**} using the first training method.*

ing units. So the interpolators were initially pretrained using the continuity model, and all the interpolators learned similar weights. We then froze the weights in the lower layers, added a small amount of noise to the interpolators' weights (uniform in $[-0.1, 0.1]$, and applied the mixture model to improve the interpolators and train the controller units. We ran the learning procedure for ten runs, each run starting from different random initial weights and proceeding for 10 conjugate gradient learning iterations. The network learned similar solutions in each case. Table 5.2 shows the value of I^{**} at the end of learning, on all ten runs.

I^{**}
14.0692963
13.8089216
14.8342957
13.1149175
14.6761029
14.6280138
13.8422717
13.7936037
12.8900913
13.3703742

Table 5.2: The final value of I^{**} after learning, on 10 runs, using the mixture of interpolators model. The value of I^{**} is summed over the interpolators for six receptive fields, and over five competing interpolators per receptive field.

A typical set of weights on one run is shown in Figure 5.10. The graph on the right in this figure shows that four of the controller units are tuned to discontinuities at different locations. The weights for the first interpolator (shown in the top left) are nearly symmetrical, and the corresponding controller's weights (shown immediately to the right) are very small; the graph on the right shows that this controller (shown as a solid line plot) mainly responds in cases when there is no discontinuity. The second interpolator (shown in the left column, second from the top) predominantly uses the leftmost three depths; the corresponding controller for this interpolator (immediately right of the top left interpolator's weights) detects discontinuities between the rightmost two depths, y_c and y_d. Similarly, the remaining controllers detect discontinuities to the right or left of y_c; each controller's corresponding interpolator uses the depths on the opposite side of the discontinuity to predict y_c.

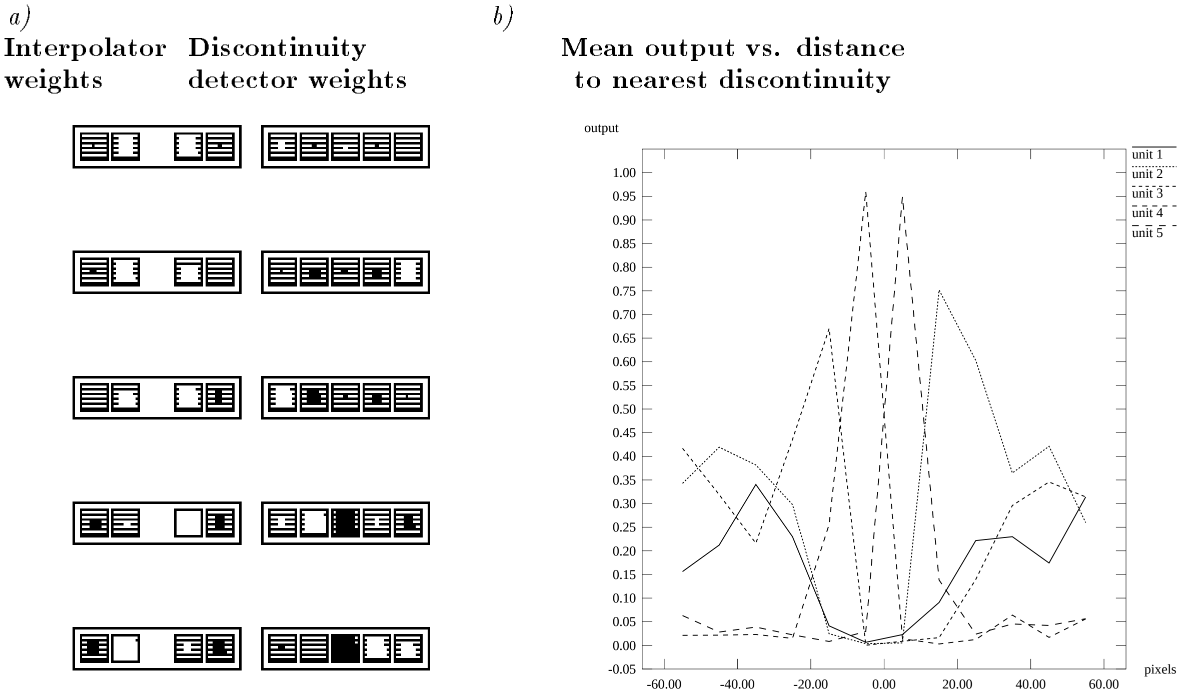

Figure 5.10: *a) Typical weights learned by the five competing interpolators and corresponding five discontinuity detectors, when trained to maximize I^{**} using the second training method. Positive weights are shown in white, and negative weights in black. b)The mean probabilities computed by each discontinuity detector are plotted against the the distance from the center of the units' receptive field to the nearest discontinuity. The probabilistic outputs are averaged over an ensemble of 1000 test cases. If the nearest discontinuity is beyond $\pm$ thirty pixels, it is outside the units' receptive field and the case is therefore a continuity example.*

5.3.1 An implementation using the competing experts framework

Provided the local depth estimates $y_a, \ldots, y_e$ do not change as the interpolators are learned (i.e., we do not back-propagate derivatives to the lower layers of the network), we can use a much simpler objective function than I^{**}. We could simply minimize a measure based on the prediction error for the interpolator $(y_c - \hat{y}_{c,i})^2$, without the possibility of the variance of y_c across cases collapsing to zero during the learning. Extending the idea to the "mixture of coherence models" case, we could fit a mixture of Gaussians model to the predicted depths y_c, adjusting the set of interpolator's weights so as to maximize the likelihood of the mixture of interpolators generating the observed depths. The re-estimation of the mixture model parameters could be done, for example, using the EM algorithm (Dempster, Laird and Rubin, 1977). A further extension is to add discontinuity detectors which compute the mixing proportions of each mixture component on each case, as in figure 5.6. This model is very close in spirit to the one in the previous section based on the I^{**} objective, and uses the same architecture (figure 5.8), but instead, maximizes the log likelihood of the model generating the data, y_c.

The model we have just described is an unsupervised version of the (supervised) competing experts algorithm described by Jacobs, Jordan, Nowlan and Hinton (1991). The output of the i^{th} expert, $\hat{y}_{c,i}$, is treated as the mean of a Gaussian distribution with variance σ^2 and the normalized output of each controller, p_i, is treated as the mixing proportion of that Gaussian. So, for each training case, the outputs of the experts and their controllers define a probability distribution that is a mixture of Gaussians. The aim of the learning is to maximize the log probability density of the desired output, y_c, under this mixture of Gaussians distribution. For a particular training case this log probability is given by:

$$\log P(y_c) = \log \sum_i p_i \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_c - \hat{y}_{c,i})^2}{2\sigma^2}\right) \tag{5.13}$$

By taking derivatives of this objective function, we can simultaneously learn the weights for the experts and the controllers. Figure 5.11 shows the weights learned by the expert interpolators and controllers on one run, using the competing experts model (maximizing equation 5.13). The network was trained for 30 conjugate gradient iterations on a set of 1000 random dot stereograms with discontinuities, using the outputs of layer 2 (after pretraining all three layers using the continuity model), as in the previous section, as input to the experts and controllers. Using this model, the network learns a perfect representation of the mixture of continuities and discontinuities, as shown in figure 5.11. There is one interpolator (the top left set of weights) that is appropriate for continuity cases and four other interpolators that are appropriate for the four different locations of a discontinuity. In ten runs from different random initial weights, the network learned virtually identical solutions.

5.4 Discussion

We have explored a number of ways of extending the (continuous) Imax algorithm to mixture models of coherence. The first extension we considered applied to the simple case where some image feature is roughly equal in neighboring image patches. In the original model from Chapter 4, when the Imax objective was applied to the outputs of pairs of neighboring modules, this led to disparity-tuned output units. Assuming a mixture model of the underlying signal, we showed that a group of competing units learn to divide up the signal space to form a population code for disparity. Such an encoding has a number of advantages. First, each unit can become finely tuned to a particular stimulus feature (e.g., a particular disparity, orientation, etc), and signal its presence with higher accuracy than it can signal a continuous range of features, assuming the unit's output has limited bandwidth. Second, the collection of units' outputs together can form a distributed representation of the parameter; this allows interpolation between units' responses, to detect a wide range of possible stimulus values. Third, we can interpret the activity across the population of output units as a probability distribution, and thereby detect cases of missing or ambiguous data: in these cases, the population should have uniformly low activity over all units. Thus, the entropy of the activity distribution represents uncertainty. Finally, a population response could signal the presence of a discontinuity in the underlying parameter, if for example, two very differently tuned units are both strongly active. Such a *a)* **Interpolator weights** **Discontinuity detector weights**

b) **Mean output vs. distance to nearest discontinuity**

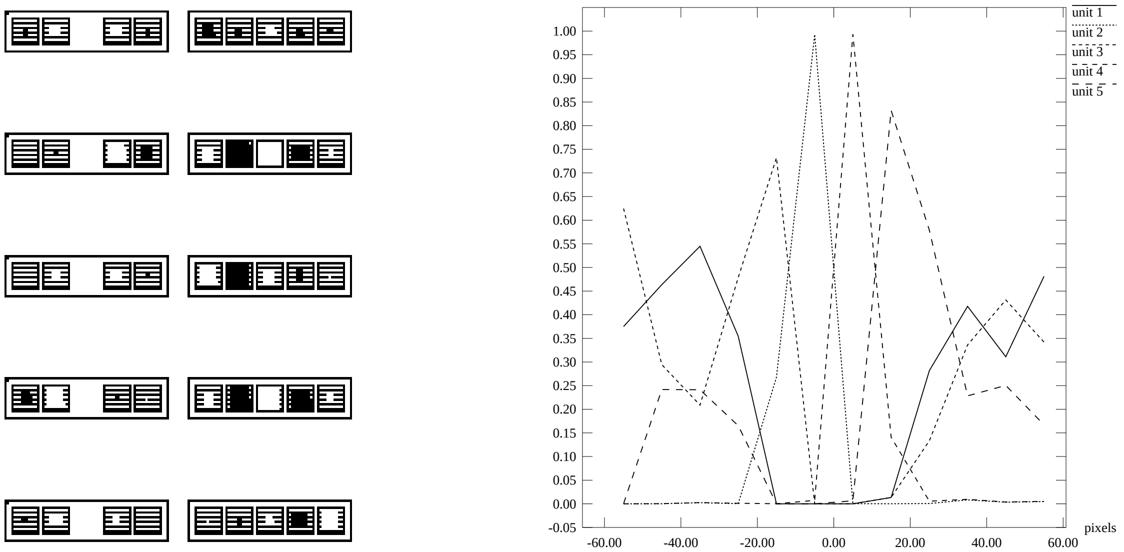

Figure 5.11: *a) Typical weights learned by the five competing interpolators and corresponding five discontinuity detectors. Positive weights are shown in white, and negative weights in black. b)The mean probabilities computed by each discontinuity detector are plotted against the the distance from the center of the units' receptive field to the nearest discontinuity. The probabilistic outputs are averaged over an ensemble of 1000 test cases. If the nearest discontinuity is beyond ± thirty pixels, it is outside the units' receptive field and the case is therefore a continuity example.*

mechanism appears to be employed in biological systems to encode parameters such as disparity (Lehky and Sejnowski, 1990).

The second extension applied in situations where an image parameter is not necessarily equal in adjacent patches, but nonetheless exhibits coherence across several neighboring patches. In our original model for this situation (from Chapter 4), a layer of depth interpolating units developed. For images with discontinuities, however, this model was inadequate. We then considered mixture models of coherence in which some cases were treated as coherent, and some as discontinuities. The first approach we explored was to simply try to discriminate between these two classes of patterns and throw away cases having discontinuities. In theory, this approach is promising, as it provides a way of making the algorithm more robust against outlying data points. Unfortunately, in order to converge to good solutions, the particular algorithm proposed here is dependent on the appropriate pre-selection of several model parameters which are data-dependent.

We then applied the idea of multiple models of coherence to a set of interpolating units, using images of curved surfaces with discontinuities. The competing controllers in figure 5.8 learned to explicitly represent which regularity applies in a particular region. The output of the controllers was used to compute a probability distribution over the various competing models of coherence. We also implemented the same model using a very different unsupervised learning procedure, based on the supervised competing experts algorithm described by Jacobs, Jordan, Nowlan and Hinton (1991). The objective function for this learning procedure is somewhat simpler than the Imax objective function, and the method produces cleaner results on the surface interpolation problem. However, the Imax objective is potentially more powerful for this problem, since it does not rely on the inputs from lower layers (the depth estimates) being fixed. One could therefore back-propagate the derivatives of the I^{**} objective function to train all layers of the network using the mixture model; this should further improve the depth-tuning of lower layers. This could not be done for the competing experts model, as the squared error for each expert could be driven to zero by always making all the depth estimates coming from lower layers zero.

The representation learned by this network has a number of advantages. We now have a measure of the probability that there is a discontinuity which is independent of the prediction error of the interpolator. So we can tell how much to trust each interpolator's estimate on each case. It should be possible to distinguish clear cases of discontinuities from cases which are simply noisy, by the entropy of the controllers' outputs. Furthermore, the controller outputs tell us not only that a discontinuity is present, but exactly where it lies. This information is important for segmenting scenes, and should be a useful representation for later stages of unsupervised learning. Like the raw depth estimates, the location of depth edges should exhibit coherence across space, at larger spatial scales. It should therefore be possible to apply the same algorithm recursively to the the outputs of the controllers, to find object boundaries in two-dimensional stereo images.

The approach presented here should be applicable to other domains which contain a mixture of alternative local regularities across space or time. For example, a rigid shape causes a linear constraint between the locations of its parts in an image, so if there are many possible shapes, there are many alternative local regularities (Zemel and Hinton, 1991).

Chapter 6

Discovering spatial coherence with Boltzmann machines

In all the models presented in this dissertation, the goal of the learning has been to extract spatially coherent parameters from nearby image patches by explicitly maximizing the mutual information between the parameters. We argued in Chapter 2 that this is a good organizing principle for perceptual systems, and may therefore have some biological significance. However, the algorithms presented so far are unlikely to be biologically plausible in terms of several of their implementation details. First, to compute the mutual information derivatives, each unit has to store a number of ensemble-averaged statistics, such as the individual and joint probabilities of units' outputs in the discrete case, and the variances of sums and differences of units' outputs in the continuous case. Second, in order to compute these statistics, units must be able to observe each other's outputs; this entails information exchange between units which are not actually linked to each other. Fourth, different modules which try to achieve high agreement must receive input from *non-overlapping* receptive fields.[1] Finally, in multi-layer networks, the learning involves propagating derivatives backwards along links in order to train the hidden layers (although in Chapter 3, we showed that the network could learn sequentially, layer by layer, to become somewhat shift-tuned).

The general idea of extracting parameters from the sensory input which are coherent across space, across time, and across different sensory modalities, may still be employed in biological systems. However, the implementation details would undoubtedly differ.

Boltzmann machines were introduced in Chapter 1, and several unsupervised approaches to training them were reviewed in Chapter 2. In this chapter, we consider an alternative class of algorithms for implementing the same learning principles in deterministic Boltzmann machines (DBMs), which avoids most of the biologically implausible details mentioned above. Results of some preliminary experiments with the model are presented for several binary shift problems. Although the algorithm is able to learn shift when the problem is linearly separable, it is only moderately successful at solving the shift problem in the general case. At the end of the chapter, we discuss reasons for this limited success, and suggest ways of improving the network's

[1] Note that if two units were directly linked to each other, they could trivially agree on every case by computing random functions of their input, and maintaining large positive weights on the connections between them. Similarly, if two units' receptive fields overlapped by a single pixel, they could trivially agree on every case by putting a large weight on this one pixel, and zero weights elsewhere.

performance.

6.1 Supervised Boltzmann machines

A Boltzmann machine (Hinton and Sejnowski, 1986) consists of a network of symmetrically connected, binary stochastic units. The probability that the ith unit adopts the "on" state, $s_i = 1$, depends on both its total input x_i and the temperature of the system, T:

$$p_i \;=\; \sigma\left(\frac{1}{T}x_i\right) = \frac{1}{1+e^{-x_i/T}} \tag{6.1}$$

where $x_i = \sum_j w_{ji}s_j$. If units are asynchronously and repeatedly updated according to the above equation, the network will eventually reach the equilibrium distribution. The network settles to a minimum of the Helmholtz free energy:

$$F = \langle E \rangle - T\ H \tag{6.2}$$

where $\langle E \rangle$ is the expected value of the energy of the system over the equilibrium distribution, and H is the entropy of this distribution. The energy of a system configuration γ is defined as follows:

$$E_\gamma = -\sum_{i<j} s_i{}^\gamma s_j{}^\gamma w_{ij} \tag{6.3}$$

At thermal equilibrium, the states of the system follow the Boltzmann distribution. Hence, the relative probabilities of two network states γ and ω in this distribution are directly related to their energies:

$$\frac{P_\gamma}{P_\omega} = \frac{e^{-E_\gamma/T}}{e^{-E_\omega/T}} = e^{-(E_\gamma - E_\omega)/T} \tag{6.4}$$

To reduce the chance of the network settling into a local energy minimum, a simulated annealing procedure is typically used, in which the system begins at high temperature, and T is gradually lowered. At each temperature, the network is permitted to settle to equilibrium.

The learning procedure for supervised Boltzmann machines (Hinton and Sejnowski, 1986) minimizes the G-error (defined in Chapter 1) between the probability distributions of output unit states given the clamped inputs in two training phases. In the positive phase, both the input and output units are clamped to particular states, I_α and O_β respectively, and in the negative phase only the inputs are clamped. The objective function to minimize is:

$$G = \sum_{\alpha,\beta} P^+(O_\beta, I_\alpha) \log \frac{P^+(O_\beta|I_\alpha)}{P^-(O_\beta|I_\alpha)} \tag{6.5}$$

where $P^-(O_\beta|I_\alpha)$ represents the probability of the output units in state β given the inputs in state α in the negative phase, when only the inputs are clamped, and $P^+(O_\beta|I_\alpha)$ represents the same probability in the positive phase, with both inputs and outputs clamped. Since this latter probability is independent of the weights, the network must minimize the following term:

$$-\sum_{\alpha,\beta} P^{+}(O_\beta, I_\alpha) \log P^{-}(O_\beta | I_\alpha) \tag{6.6}$$

The appealing aspect of the learning procedure is the simple form of the weight update rule derived by differentiating G with respect to the weights (Hinton and Sejnowski, 1986):

$$\frac{\partial G}{\partial w_{ji}} = -\frac{1}{T}\left[\langle s_i s_j \rangle^{+} - \langle s_i s_j \rangle^{-}\right] \tag{6.7}$$

where $\langle s_i s_j \rangle^{+}$ denotes the expectation of the product of the states of the ith and jth units, averaged over all training cases, and over all system states in the equilibrium distribution for each training case, in the positive phase (when the both input and output units are clamped), and where $\langle s_i s_j \rangle^{-}$ denotes the expectation of the same quantity in the negative phase, when only the inputs are clamped. Note that this "contrastive Hebbian rule" depends only on information locally available to each connection. Thus, the learning procedure can be applied to multi-layer recurrent networks and requires no back-propagation of derivatives.

The unappealing aspect of the Boltzmann learning procedure is its slowness, due largely to the stochastic sampling required to accurately estimate the correlations $\langle s_i s_j \rangle$ in the two phases. The deterministic Boltzmann machine (DBM) (Peterson and Anderson, 1987) overcomes this problem by using the mean field approximation to compute the correlations:

$$\langle s_i s_j \rangle \tilde{=} \langle s_i \rangle \langle s_j \rangle \tag{6.8}$$

This approximation is obtained under the maximum entropy assumption, in which the fluctuations in the units' states at equilibrium (for a particular training case) are assumed to be independent. Using this approximation, we can replace the stochastic state update function of equation 6.1 with a deterministic state update function, in which the real-valued state of a unit is equal to the expected value of the stochastic function:

$$\begin{aligned} p_i &= \langle s_i \rangle \\ &= \left\langle \sigma\left(\frac{1}{T}\sum_j w_{ij} s_j\right) \right\rangle \\ &\tilde{=} \left\langle \sigma\left(\frac{1}{T}\sum_j w_{ij} p_j\right) \right\rangle \end{aligned} \tag{6.9}$$

We can update the states p_i using a synchronous, discrete time approximation to the differential equations:

$$\frac{\partial p_i}{\partial t} = -p_i + \frac{1}{1 + e^{-x_i/T}} \tag{6.10}$$

The following approximation is typically used:

$$p_i(t) = \eta p_i(t-1) + (1-\eta)\frac{1}{1 + e^{-x_i(t)/T}} \tag{6.11}$$

where $0 < \eta < 1$.

In a stochastic Boltzmann machine, there is a simple relationship between the relative probabilities of two distributions and their free energies. For example, we can compute the relative probability of an input pattern I_α with the outputs clamped and unclamped, since the following equality holds at equilibrium (Hinton, 1989):

$$\frac{P(I_\alpha, O_\beta)}{P(I_\alpha)} = \frac{e^{-F^*_{\alpha\beta}/T}}{e^{-F^*_\alpha/T}} \tag{6.12}$$

where $F^*_{\alpha\beta}$ is the free energy minimum of the distribution with I_α and O_β clamped, and F^*_α is the minimum with just I_α clamped.

In the deterministic case, we can no longer observe the relative probabilities of particular binary states of the network. However, the free energies of the "mean field" distributions represented by the network for each input pattern can be computed as follows, under the mean field approximation:

$$\begin{aligned} F &= \langle E \rangle - T\,H \\ &\doteq -\sum_{i<j} p_i p_j w_{ji} + T\left[\sum_j p_j \log p_j + (1-p_j)\log(1-p_j)\right] \end{aligned} \tag{6.13}$$

We can arbitrarily define the relative probabilities of distributions of states represented by the DBM using equation 6.12, but substituting the free energy minima of the particular states that the DBM settles into for the free energies of the distributions for the stochastic BM. Using this definition of the probabilities of DBM states, and the maximum entropy assumption, Hinton (1989) has shown that the same learning rule used for training stochastic Boltzmann machines can be applied to DBMs, and that it performs gradient descent in the same objective function as does the stochastic Boltzmann machine learning rule:

$$-\sum_{\alpha\beta} P^+(O_\beta, I_\alpha) \log P^-(O_\beta | I_\alpha) \tag{6.14}$$

The learning rule for each weight on each case is obtained by replacing the correlational statistics by their mean field approximations:

$$\Delta w_{ji} = \epsilon(p_i^+ p_j^+ - p_i^- p_j^-) \tag{6.15}$$

6.2 Unsupervised Boltzmann machines

The Boltzmann machine can be trained in an unsupervised manner by minimizing the G-error between probabilities of visible unit states in the clamped and unclamped phases. This objective function is now equivalent to maximizing the log probability of the network generating the patterns in the training set. However, for a network of the same size (replacing the input units in the supervised Boltzmann machine by unclamped hidden units) it is now much more complicated to compute this probability. From equation 6.4, the absolute probability of a single state γ can be computed by comparing the energy of that state to the energies of all other possible network states:

$$\sum_{\omega} \frac{P_{\omega}}{P_{\gamma}} = \sum_{\omega} \frac{e^{-E_{\omega}/T}}{e^{-E_{\gamma}/T}} \tag{6.16}$$

$$P_{\gamma} = \frac{e^{-E_{\gamma}/T}}{\sum_{\omega} e^{-E_{\omega}/T}} \tag{6.17}$$

Unfortunately, it is rather expensive to compute the partition function (the denominator in equation 6.17). We have to sum over all possible unclamped unit states, of which there are 2^n for n units.

Another drawback of this type of unsupervised learning is that the mean field approximation is not particularly useful in this case. A DBM cannot form an adequate representation of the unclamped distribution. All 2^n states of the n units in this distribution are represented by a single mean state. Clearly, this representation results in a loss of information. The information lost is in the pairwise and higher order probabilities of units being on together.

6.3 Learning by contrastive clamping

We can improve the complexity of unsupervised Boltzmann machine learning considerably by using an alternative training method, which is particularly well suited to the problem domain of learning spatially coherent features in images. Instead of maximizing the *absolute* probability of network configurations when the inputs are clamped to spatially coherent patterns, we can maximize the *relative* probability of these configurations with respect to other carefully selected patterns. The training set will consist of a mixture of coherent and incoherent patterns. Each training pattern is labelled as either being coherent or incoherent. By giving the network this 1 bit of "supervised" information with every pattern, it should be able to learn to model the difference between the two classes of patterns.

The objective is to maximize the sum of the log probabilities of network's distributions of states on positive (coherent) cases, relative to the overall probability of network states for the entire training set (which includes both positive and negative cases). We can compute the relative probability of a network state when the inputs are clamped to a positive pattern, α, as follows:

$$\frac{P(I_{\alpha})}{\sum_{\beta} P(I_{\beta})} = \frac{e^{-F^*_{\alpha}/T}}{\sum_{\beta} e^{-F^*_{\beta}/T}} \tag{6.18}$$

where β indexes over both positive and negative cases. This objective applies equally well to the DBM case, using the mean field approximation described in the previous section.

The derivative of the objective function, the relative log likelihood of the positive training cases, with respect to one of the weights in the network, w_{ji}, for a (positive) training case α, can be computed as follows (see Appendix D for the full derivation):

$$\frac{\partial \log\left(\frac{P(I_{\alpha})}{\sum_{\beta} P(I_{\beta})}\right)}{\partial w_{ji}} = \frac{1}{T}\left[p_i^{\alpha} p_j^{\alpha} - \sum_{\beta} \frac{P(I_{\beta})}{\sum_{\delta} P(I_{\delta})} p_i^{\beta} p_j^{\beta}\right] \tag{6.19}$$

where β and δ index over all cases, both positive and negative, in the training set. Thus, we still have a contrastive Hebb-like learning rule, except that each negative term must be weighted by the relative probability of the corresponding case in the training set. This probability can be computed from equation 6.18, by observing the equilibrium free energy of the network on each case, using the mean field approximation in equation 6.15.

We have applied this learning procedure to a synthetic binary stereo problem, as described in Chapter 3 for the discrete version of Imax. We create an ensemble of binary shift patterns which are divided into positive and negative cases. In the positive cases, the shift between the left and right halves of the input pattern is coherent across space, as in the input pattern of figure 6.1a). In the negative cases, the shift changes abruptly across different parts of the input, as in figure 6.1b).

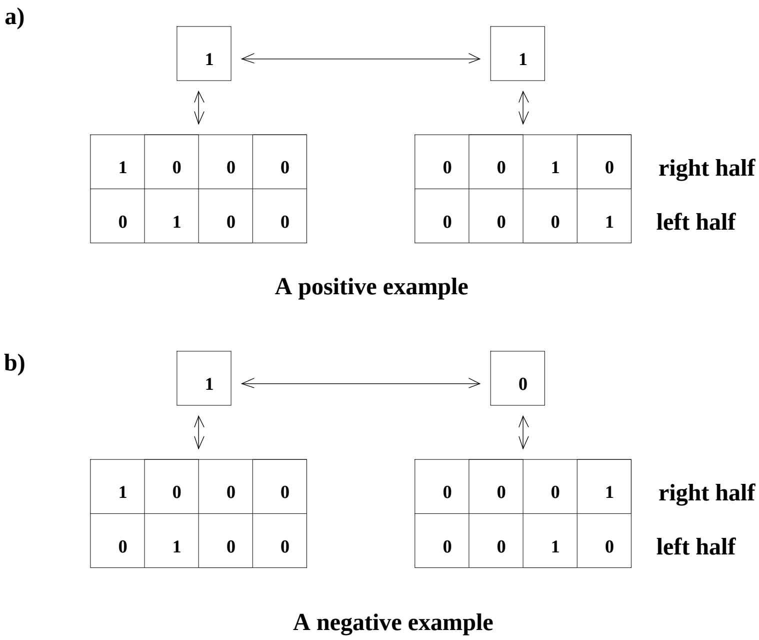

Figure 6.1: *A Boltzmann machine that learns by "contrastive clamping". The network consists of two modules with symmetrically connected output units, each connected to different parts of the input. In the positive phase, the two parts of the input are clamped to a spatially coherent pattern in which the shift between the left and right halves is the same for both receptive fields, as shown in a) above. In the negative phase, the shift varies across different parts of the input, as in b) above.*

We can use the same basic architecture for the above problem that we used for Imax, as shown in figure 6.1. The input layer is divided into two receptive fields, and each output unit receives input from one of the receptive fields. Additionally, the output units are directly connected to each other. In order to maximize the log relative probability (or equivalently, to minimize the relative free energy) of configurations on positive examples, the network should learn to positively weight the connection between the two outputs, and try to make the output units agree on positive cases and disagree on negative examples (or the converse, with a negative connection between the outputs). When shown positive patterns like those in figure 6.1a), the two output units should adopt the same states, and when shown negative examples like those in figure figure 6.1b), they should adopt opposite states. To achieve this solution, the network must learn to extract the shift, for example, by turning the outputs both on for coherent right shifts, both off for coherent left shifts, and one on and one off for incoherent shifts.

6.3.1 Using n-valued codes to learn multiple shifts

An unsupervised Boltzmann machine consisting of binary state units can learn to extract shift, to some degree, when there are only two possible shifts in the data (Nowlan, 1990). For patterns with multiple shifts, we can apply a generalization of the state update rule in equation 6.11 to n-state units. An n-state unit can be simulated by a set of units whose outputs are positive, and normalized to sum to 1:

$$p_i = \frac{e^{x_i/T}}{\sum_j e^{x_j/T}} \tag{6.20}$$

Hopfield and Tank (1985) used this generalization in Hopfield networks, and Peterson and Söderberg (1989) have applied it to DBMs.

Under this new model, using the mean field approximation, the free energy of a configuration at equilibrium is:

$$\begin{aligned} F &= \langle E \rangle - T\ H \\ &\tilde{=} -\sum_{i<j} p_i p_j w_{ji} + T \sum_j p_j \log p_j \end{aligned} \tag{6.21}$$

and the learning rule is identical to that of equation 6.19 (derived in Appendix D). All the experiments described in the remainder of this chapter use the model described in this subsection.

6.3.2 Pretraining without settling

The advantage of allowing the DBM to settle to equilibrium on each learning iteration is in the simple form of the learning rule. At equilibrium, $\frac{\partial F}{\partial p_i} = 0, \forall p_i$, so $\frac{\partial F}{\partial w_{ji}} = -p_i p_j$. However, the settling time slows down the learning considerably. To speed up the learning, an alternative training method is to not do any settling, and compute the derivatives for the non-equilibrium case. Unfortunately, there is no longer a correspondence between probability distributions and free energies, as given by equation 6.12. However, we can still maximize the right hand side of this equation:

$$\frac{e^{-F_\alpha/T}}{\sum_\beta e^{-F_\beta/T}} \tag{6.22}$$

with the free energy of the distribution for input case α defined as before, using equation 6.21.

There still remains one problem, which is how to update the states of the network without doing any settling, given that it has recurrent connections. The networks we will use for this learning procedure have highly restricted connectivity, like the ones shown in figure 6.2. Once the input units are clamped to a pattern, the connections from input to output units effectively operate as feed-forward links. So the only effective feedback is along links between the output units. For a non-settling network, we can simply initialize the states of the output units to zero, and only propagate states for one iteration. So no activity spreads across these feedback links, although the weights on these links still contribute to the free energy of the system. We can now use a non-iterative version of the state update equation for multi-state units:

$$p_i = \frac{e^{x_i/T}}{\sum_j e^{x_j/T}} \tag{6.23}$$

The derivatives of equation 6.22 are much more complicated for a non-settling network with multi-state units. The derivations of the learning equations are given in Appendix D. Although this learning procedure is not as conceptually clean as for the settling DBM, it performs as well in practice, and is much faster. It is therefore useful for the purpose of our experiments.

6.4 Experiments with single-layer networks

Preliminary experiments with linearly separable patterns demonstrate that when a single-layer network is trained with the contrastive clamping procedure described, it is able to separate the positive and negative cases. Further, with multiple output units per module, the units learn to divide up the shifts. We trained the single layer networks shown in figure 6.2a) and b) on very simple shift patterns.

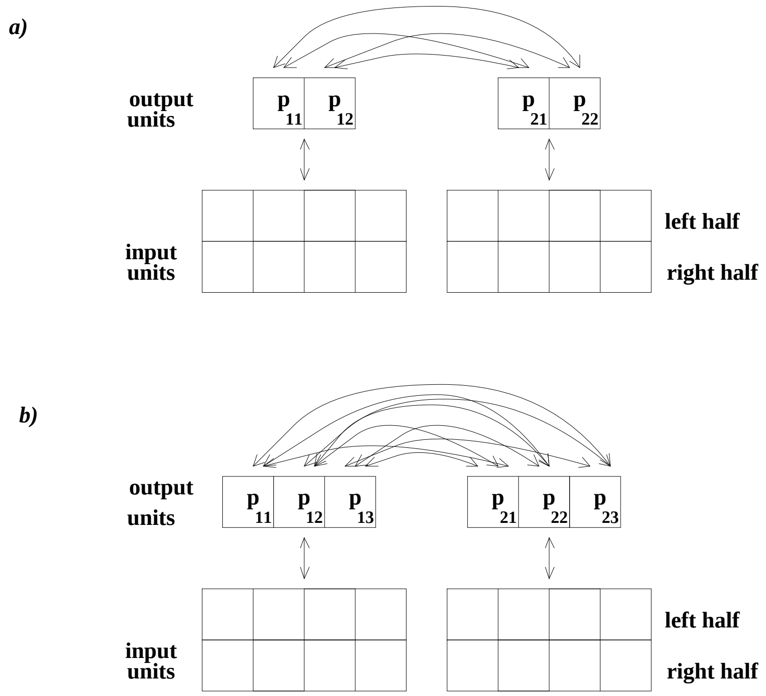

Figure 6.2: *Two network architectures used for discovering multiple shifts in binary patterns, in unsupervised DBMs. The network in a) has two output units per module, and can learn to extract two shifts. The network in b) has three outputs per module, and can learn to extract three shifts. The output units of each module are not linked together, but rather, their activities are normalized, in order to produce a "1-of-n encoding".*

The patterns were divided into two parts, each part corresponding to the receptive field of one of the DBM modules. Each part of the pattern consisted of a right half and a left half. The right half contained one of four possible binary strings having a single bit on: 1000, 0100,0010, or 0001. The left half contained a shifted version of the right half, using shift *without wrap-around*. The data sets had either two or three possible shifts, left and right shifted patterns in the former case, with the addition of unshifted patterns in the latter case. So, for the 3-way shift pattern ensemble, the set of twelve possible subpatterns seen by each module were:

$$\begin{aligned}
L &= \left\{ \begin{matrix}1000\\0000\end{matrix}, \begin{matrix}0100\\1000\end{matrix}, \begin{matrix}0010\\0100\end{matrix}, \begin{matrix}0001\\0010\end{matrix} \right\} \\
R &= \left\{ \begin{matrix}1000\\0100\end{matrix}, \begin{matrix}0100\\0010\end{matrix}, \begin{matrix}0010\\0001\end{matrix}, \begin{matrix}0001\\0000\end{matrix} \right\} \\
N &= \left\{ \begin{matrix}1000\\1000\end{matrix}, \begin{matrix}0100\\0100\end{matrix}, \begin{matrix}0010\\0010\end{matrix}, \begin{matrix}0001\\0001\end{matrix} \right\}
\end{aligned}$$

A complete set of 64 patterns for the entire network (both modules), for the two-way shift problem, consisted of the union of all possible pairwise cross-products of the L and R sub-pattern sets, examples of which are shown in figure 6.1. For the three-way shift problem a complete set of 144 patterns consisted of the union of all cross-products of the L, R and N sub-pattern sets. Positive examples consisted of the union of the sets $L \times L$, $R \times R$, and additionally for the 3-way shift problem, $N \times N$. Negative examples consisted of the union of the sets $L \times R$, $R \times L$, and additionally for the 3-way shift problem, $L \times N$, $N \times L$, $R \times N$, and $N \times R$.

In preliminary experiments with the two-way shift problem, using the network shown in figure 6.2a), we compared the network performance when the network was trained with the settling and non-settling methods. For the settling method, on each pattern presentation, simulated annealing was performed with an initial temperature of 40 and a final temperature of 1. At each temperature, states were iteratively, synchronously updated until the network reached equilibrium, using the following state update equation:

$$p_i(t) = \eta p_i(t-1) + (1-\eta)\frac{e^{x_i(t)/T}}{\sum_j e^{x_j(t)/T}} \tag{6.24}$$

where t indexes the iteration, and where $\eta = 0.5$. The network was considered to have reached equilibrium when no state changed by more than 0.001 on the same iteration. After each iteration, the temperature was decayed by a factor of 0.92. The non-settling network used the state update function given in equation 6.23.

While the network was able to solve the two-way shift problem virtually perfectly using both the settling and non-settling methods (we explain below what we mean by "perfect performance"), the training with the settling method of course took much longer because of the annealing time on each case. Both methods took about the same number of learning iterations to converge, using the method of conjugate gradients. In fact, the settling is not necessary to solve this particular problem, on the data set described above. If, however, we were to use a data set containing some cases with no data, and the network was required to interpolate across the receptive fields of several modules in order to solve the problem, then the lateral links would be necessary to propagate states across modules. As the present problem does not require interpolation, the only purpose served by the lateral weights is to encode the positive or negative correlations between output units, so as to improve the objective function.

When the network was trained first with the non-settling method, followed by several learning iterations with the settling method, we found that the subsequent training with settling did not change the solution much, except that the weights on the lateral connections between the output units of the two modules tended to shrink. The non-settling training method tended to produce very large weights on these lateral