# Chapter 6

# A Neural Network for Computing Invariance Signatures

## 6.1 The Invariance Signature Neural Network Classifier

We propose a Model-Based Neural Network to compute the discrete Invariance Signature of an input pattern, as described in §5.2.4, and to classify the pattern on that basis. This MBNN consists of a fairly complex system of neural network modules, some hand-coded and some trained on sub-tasks. A schematic diagram is shown in Figure 6.1. This system, first described in Squire and Caelli (1995), will be referred to as the Invariance Signature Neural Network Classifier (ISNNC).

Whilst the ISNNC appears complex, it retains the basic characteristics of a traditional feed-forward neural network. It consists entirely of simple nodes joined by weighted connections.[1] Each node $i$ in the network computes the sum of its $j$ weighted inputs, $net_i$,

$$net_i = \sum_j w_{ij} x_j. \tag{6.1}$$

This is then used as the input to the transfer function $f$ of the node, which is either linear, $f(net_i) = net_i$, or the standard sigmoid,

$$f(net_i) = \frac{1}{1 + e^{-net_i}}. \tag{6.2}$$

The only departure from a traditional neural network at runtime is that some of the

[1] with the exception of the *Dominant Image Orientation Unit* for which a neural network solution is still to be developed.

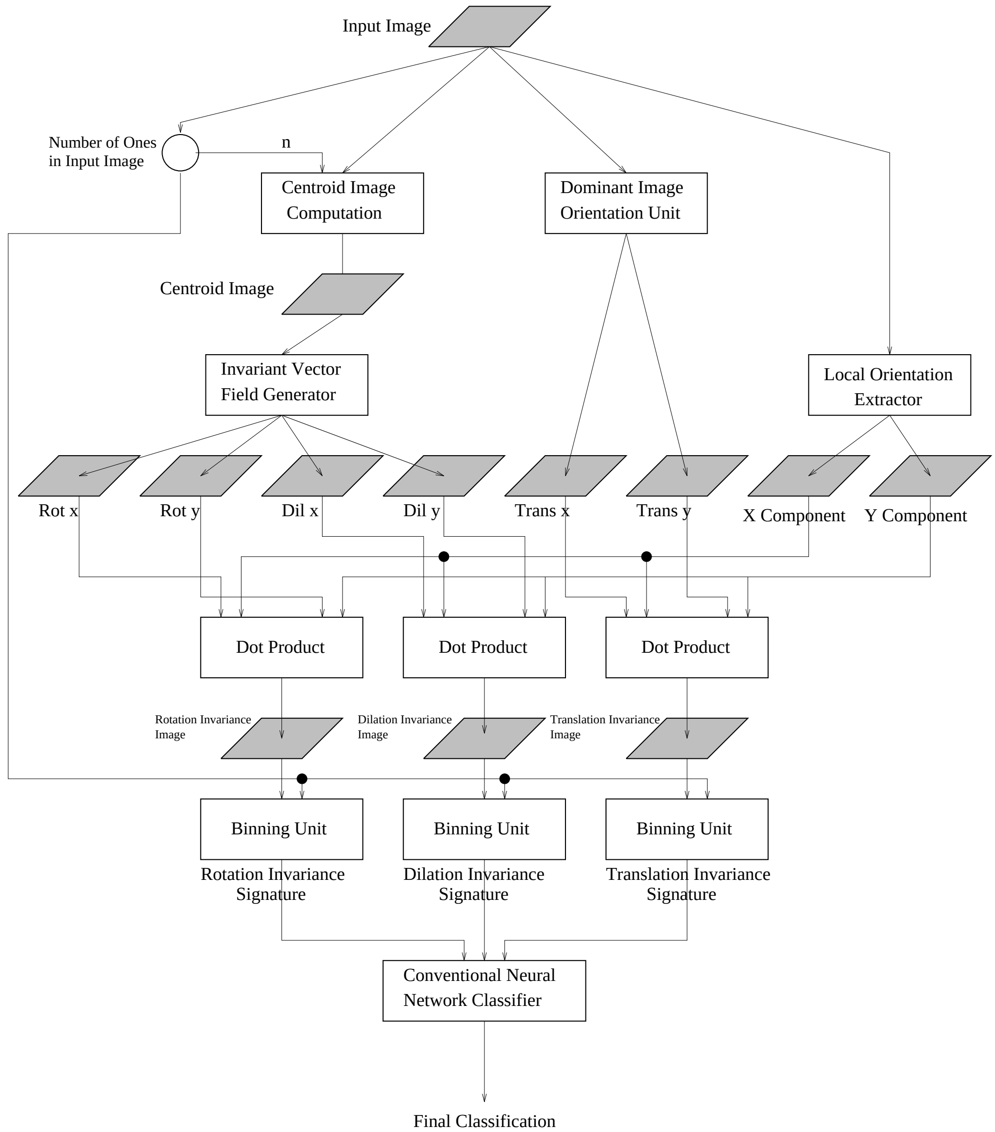



Figure 6.1: Invariance Signature-based contour recognition system.

connection weights are calculated by a node higher up in the network. We call these *dynamic weights*. The presence of dynamic weights allows the ISNNC to compute dot products,[2] and also for some nodes to act as gates controlling whether or not the output of a node is transmitted. Since connection weights in any implementation of a neural network are no more than references to some stored value, this should not present any difficulty.[3] Alternatively, the same functionality can be achieved by allowing a new class of node which multiplies (possibly weighted) inputs. Nodes of this type are

[2]The calculation of dot products, for example, is achieved by having the outputs of one layer provide the *weights* on connections to another layer.

[3]The aim of our research with MBNNs has never been biological plausibility. Perhaps this runtime weight calculation can be thought of as real-time, rapid, accurate training.

a component, for instance, of Higher Order Networks [Perantonis and Lisboa, 1992; Redding et al., 1993; Schmidt and Davis, 1993; Spirkovska and Reid, 1994; Delopoulos et al., 1994].

The ISNNC consists of a number of distinct levels. Data is processed in a feed-forward manner, and computation at a given level must be completed before computation at the next level can begin. Each of the following sections describes computation at a particular level.

### 6.1.1 Lie Vector Field Generation

#### Calculation of a Centroid Image

The first step in creating the vector fields corresponding to the generators of the rotation and dilation groups is to compute the coordinates of the centroid of the image, since, as seen in §5.2.2, this must be the origin of coordinates. This is done using a neural module which takes as its input a binary image, and outputs an image of the same size which is zero everywhere except at the centroid, where it is one. In Figure 6.1 this is labeled "Centroid Image". The weights of the Centroid Image Computation module are entirely hand-coded.

A quantity needed for this operation, and also later in the ISNNC, is $N_{\text{on}}$, the total number of "on" pixels in the input binary image $\mathcal{I}(x_i, y_j)$. This is given by

$$N_{\text{on}} = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} \mathcal{I}(x_i, y_j). \tag{6.3}$$

This can easily be computed by a single node with a linear transfer function which has inputs from all input nodes with weights of 1. In fact, the value used in later levels of the ISNNC is $\frac{1}{N_{\text{on}}}$. Since the size of the input image is fixed, it is trivial to build a neural module which interpolates the function $f(x) = \frac{1}{x}$ with any desired accuracy for the integer values $\{x : x \in [0, N_x N_y]\}$. Bulsari (1993), for instance, demonstrates how to construct piecewise constant approximations to problems such as this with neural networks. From now on it will be assumed that the value $\frac{1}{N_{\text{on}}}$ is available.

The equations for the coordinates, $(\bar{x}, \bar{y})$, of the centroid of $\mathcal{I}(x_i, y_j)$ are

$$\begin{aligned} \bar{x} &= \frac{1}{N_{\text{on}}} \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} \mathcal{I}(x_i, y_j)\, x_i \\ \bar{y} &= \frac{1}{N_{\text{on}}} \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} \mathcal{I}(x_i, y_j)\, y_i \end{aligned} \tag{6.4}$$

A simple system of two linear neurons can compute each of these quantities. Such a module is shown in Figure 6.2. This illustrates the notion of dynamic weights mentioned above. The weight on the single connection between Node A and Node B is dynamic, being computed from the number of ones in the input image. The weights on the connections to Node A are the $x$-coordinates of each of the input nodes.[4] These weights are fixed during the design process, so there is no sense in which a node needs to "know" what its coordinates are. An identical system is used to compute $\bar{y}$.

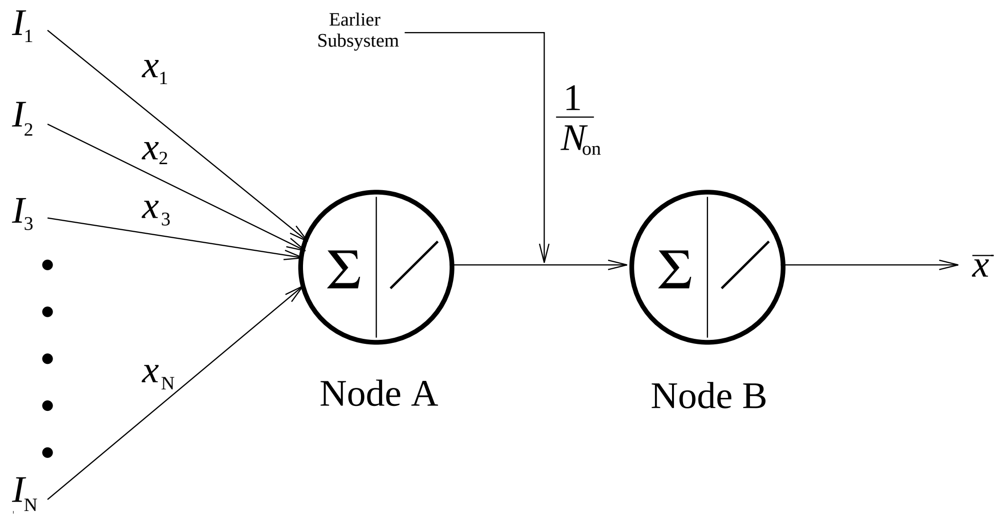



Figure 6.2: $\bar{x}$ module.

Once $\bar{x}$ and $\bar{y}$ have been calculated, the Centroid Image can be generated. This is done using a pair of nodes for each input node, as shown in Figure 6.3.[5] The weights and inputs to Nodes A and B are set so that they only both "fire" if their coordinate is within $\theta$ of the centroid coordinate. Here $\theta = 0.5$ was used, since the input nodes were assigned unit spacing. A neural AND of these nodes is then calculated by Node C.

This results in two images, one for the $x$ coordinate and one for $y$. Each has a straight line of activation corresponding to the centroid value for that coordinate. A neural AND of these images is computed, giving the final output Centroid Image, $\mathcal{I}_C$, in which only the centroid node is one.

### Gating of Vector Fields

Once the Centroid Image has been calculated, it is used to produce the rotation and dilation vector fields, four images in all. The weights from the Centroid Image are all derived from equations 5.50 and 5.51. Each centroid image node has weights going to all the (linear) nodes in each vector field component layer. Each weight has the

[4]Not to be confused with their output values, which are also often denoted as $x_i$ or $y_i$.

[5]The parameter $\alpha$ in these figures effectively determines the gain of the sigmoids, since it scales all weights. In this study it was set to 100.

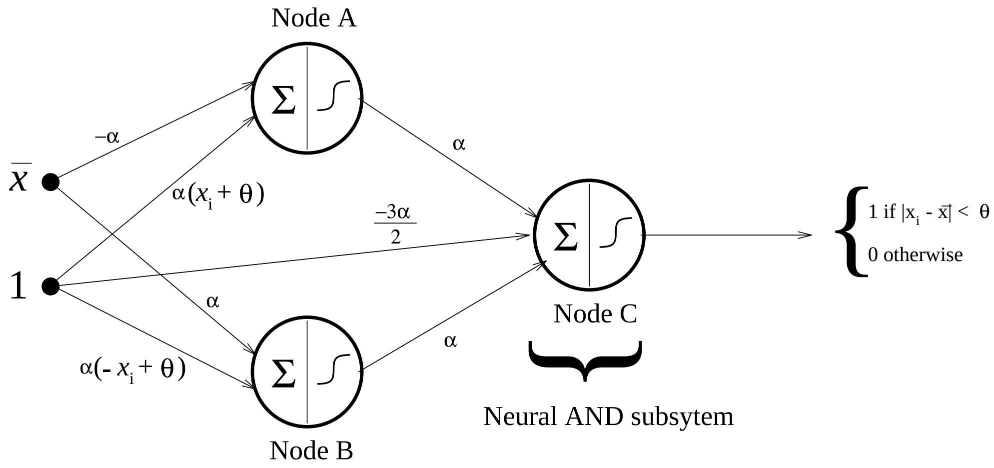



Figure 6.3: Coordinate matching module for node $x_i$. Output 1 when: $|x_i - \bar{x}| < \theta$.

value appropriate for the vector field at the destination node if the source node is the centroid. Since all nodes of the Centroid Image are zero except for the centroid node which is one, only the weights from the centroid node contribute to the vector field component images. Thus, weights corresponding to the vector fields for all possible centroid positions are present in the network, but only the appropriate ones contribute. Equation 6.5 shows the function computed by a node $(k, l)$ in the rotation invariance vector field $x$-component image:

$$\begin{aligned}
\mathrm{Rot}_{x_{(k,l)}} &= \sum_{i=0}^{x_{max}} \sum_{j=0}^{y_{max}} w_{(i,j)(k,l)} \mathcal{I}_{C_{(i,j)}} \\
w_{(i,j)(k,l)} &= \frac{-(j-l)}{\sqrt{(j-l)^2 + (i-k)^2}}.
\end{aligned} \tag{6.5}$$

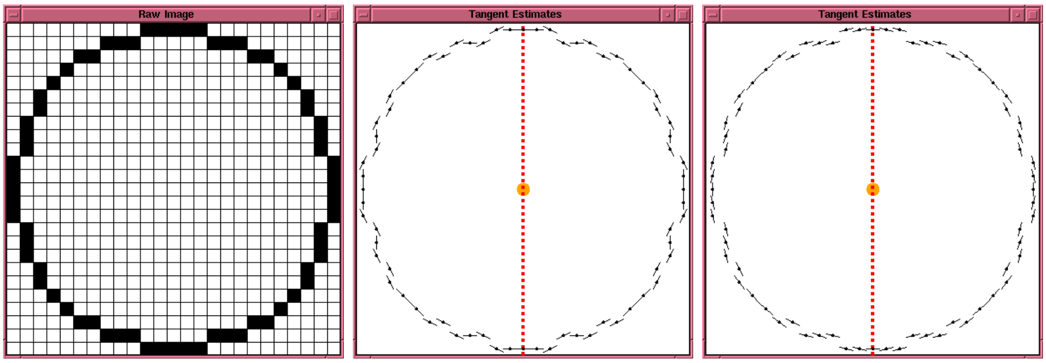
Note that it does not matter which node is chosen as the origin of coordinates in each layer when the weights are set, only that it is consistent between layers. Similar functions for setting weight values, derived from Equations 5.50 and 5.51, are used for the other three vector component images.

### 6.1.2 Local Orientation Extraction

#### A Tangent Estimate

In order to compute the Invariance Signature for an image, it is necessary to estimate the tangent vector at each point of the contour, since the mapping to invariance space requires the dot product of this with the three vector fields described in §5.2. A simple and robust estimate of the tangent vector at a point is the eigenvector corresponding to the largest eigenvalue of the covariance matrix of a square window centred on that point.

The size of the window chosen defines the number of orientations possible. Figure 6.4 shows the orientation of the tangent estimates for an image of a circle for both $3 \times 3$ and $5 \times 5$ windows, calculated exactly. It is clear that the tangent estimates




(a) Original Image (b) $3 \times 3$ Window (c) $5 \times 5$ Window

Figure 6.4: Tangent estimation with varying window sizes, using the eigenvector corresponding to the largest eigenvalue of the covariance matrix.

calculated using the $5 \times 5$ window are better than those from the $3 \times 3$, but this comes at a computational cost, since the number of calculations is proportional to the square of the window size. Moreover, the "best" choice of window size depends upon the scale of the input contour. If the window size becomes large compared to the scale of the contour, the dominant eigenvector of the window's covariance matrix becomes a poor estimator of the tangent at a point, since the window is likely to include extraneous parts of the contour. This is clear from the tangent estimates shown in Figure 6.5, which was calculated with a window size of $35 \times 35$. Consequently, we choose to use a $3 \times 3$ window, both for its computational advantage and because it makes no assumptions about the scale of the input contour.

### Discontinuities

Estimating this mapping presents some difficulties for a neural network, as it is not continuous. There are two discontinuities. The first arises when the dominant eigenvalue changes, and the orientation of the tangent estimate jumps by $\frac{\pi}{2}$ radians. The second is due to the flip in sign of the tangent vector when the orientation goes from $\pi$ back to 0.

The first discontinuity can be avoided by using a weighted tangent estimate. Rather than being unity, we let the magnitude of the estimated tangent vector corresponds to strength of the orientation. Let the eigenvalues of the covariance matrix be $\lambda_1$ and

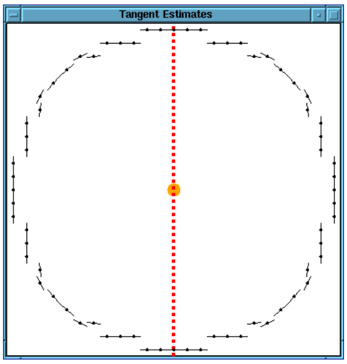



Figure 6.5: Tangents estimated with a window size of $35 \times 35$.

$\lambda_2$, where $\lambda_1 \geq \lambda_2$. The corresponding unit eigenvectors are $\hat{\mathbf{e}}_1$ and $\hat{\mathbf{e}}_2$. The weighted tangent vector estimate $\mathbf{s}$ is given by

$$\mathbf{s} = \begin{cases} \left(1 - \left|\frac{\lambda_2}{\lambda_1}\right|\right) \hat{\mathbf{e}}_1 & \text{if } \lambda_1 \neq 0, \\ 0 & \text{if } \lambda_1 = 0. \end{cases} \tag{6.6}$$

This weighting cause the magnitude of the estimated tangent vector to go to zero as $\left|\frac{\lambda_2}{\lambda_1}\right| \rightarrow 1$, and thus the $\frac{\pi}{2}$ jump is avoided. There is, however, still a discontinuity in the derivative.

### Training a Neural Orientation Extraction Module (NOEM)

The aim, then, is to produce a neural network module to approximate this mapping. This module is to be replicated across the input layer so that the $x$ and $y$ components of the tangent vector are estimated everywhere. The result is two images of the same size as the input layer. The values are gated by the input image (used as dynamic weights), so that values are only transmitted for nodes corresponding to 1's in the input image. It was decided to produce this module by training a network on this task.

A training set was produced consisting of all 256 possible binary $3 \times 3$ input windows with a centre value of 1, and, for each, the two outputs given by Equations 6.6. A variety of networks was produced and trained on this task. The performance measure used, $E$, was the ratio of the sum squared error to a variance measure for the training set,

$$E = \frac{\sum_{c=1}^{N} \sum_{j=1}^{M} \left(t_{cj} - y_{cj}\right)^2}{\sum_{c=1}^{N} \sum_{j=1}^{M} \left(\overline{t_{cj}} - t_{cj}\right)^2} \tag{6.7}$$

where $N$ is the number of training exemplars, $M$ the number of nodes in the output layer, $t_{cj}$ the desired output value of the node and $y_{cj}$ the actual output value.

A variety of standard, single hidden layer backpropagation networks was trained, with different hidden layer sizes. It was found that the task could not be learnt without a significant number of hidden nodes, which is not unexpected for such a highly nonlinear function. The networks proved very unstable during training, so it was necessary to use an extremely small learning rate. As would be expected, the accuracy of the mapping improved with increasing hidden layer size, and the training time increased. The residual error did not stabilize, but continued to decrease steadily, albeit very slowly, as training was continued. The final accuracy achieved was thus ultimately determined by how long one was prepared to wait.

For the work reported in Squire and Caelli (1995), a three layer neural network with a $3 \times 3$ input layer, a 20 node hidden layer, and 2 output nodes was trained using backpropagation. After $6 \times 10^6$ iterations with a learning rate of 0.0005, a value of $E = 3.11\%$ was reached. This was considered sufficiently accurate for this module. Further investigations, as will be seen in §7.2, have indicated that even such seemingly small residual errors in the Local Orientation Extraction module can adversely affect final classification performance. Consequently, means of improving the accuracy of this module were sought.

Despite the universality of single hidden layer neural networks, several authors have indicated that networks with multiple hidden layers can be useful for extremely nonlinear problems [Sontag, 1992; Sarle, 1994]. Sarle (1994, p. 9) reports:

> Although a MLP with one hidden layer is a universal approximator, there exist various applications in which more than one hidden layer can be useful. Sometimes a highly nonlinear function can be approximated with fewer weights when multiple hidden layers are used than when only one hidden layer is used.

This suggested that a more compact solution might be obtained using a network with two hidden layers. Several standard four-layer networks were trained using backpropagation on this problem. It did indeed seem that they converged more rapidly, and to a smaller residual error. However, as with the single hidden layer networks, they were very sensitive to the learning rate, suggesting that perhaps a more sophisticated learning algorithm, such as QuickProp [Fahlman, 1988], would be more appropriate. After an initial large reduction, the error continued to decrease very slowly throughout training. The limitation on the final accuracy of the approximation appeared to be how long one was prepared to wait as much as the number of hidden nodes.

It is noted that this is a similar problem to edge extraction, the difference being that edge extraction is usually performed on images containing grey-scale gradients rather than on a binary, thin contour. Srinivasan, Bhatia and Ong (1994) have developed a neural network edge detector which produces a weighted vector output very much like the one described in Equation 6.6. They used an encoder stage which was trained competitively, followed by a backpropagation-trained stage. The encoder produced weight distributions closely resembling Gabor filters of various orientations and phases.

It is intended to try to produce a more compact and accurate tangent estimator using a MBNN incorporating a stage with Gabor weighting functions, as used in Caelli et al. (1993).

## 6.2 Calculation of the Local Measure of Consistency

The next stage of the network computes the Local Measure of Consistency of the tangent estimate at each point of the contour for each of the Lie vector fields. The output required is an *Invariance Image* for each Lie transformation. This is an image of the same size as the input image, where the value at each point is given by the absolute value of the dot produce of the estimated tangent vector to the contour and the Lie vector field at that point, as specified in Equation 5.34.

The neural implementation is simple. With reference to Figure 6.1, the $x$-component image from the Local Orientation Extractor provides dynamic weights to be combined with the $x$-component of each of the Lie vector fields. The same is done for the $y$-components. For each Lie transformation, there is thus layer of neurons which each have two inputs. The inputs come from the vector field images at the same coordinates, and are weighted by the Local Orientation images, again at the same coordinates. Note that points corresponding to zeros in the input image have tangent estimates of zero magnitude and thus make no contribution.

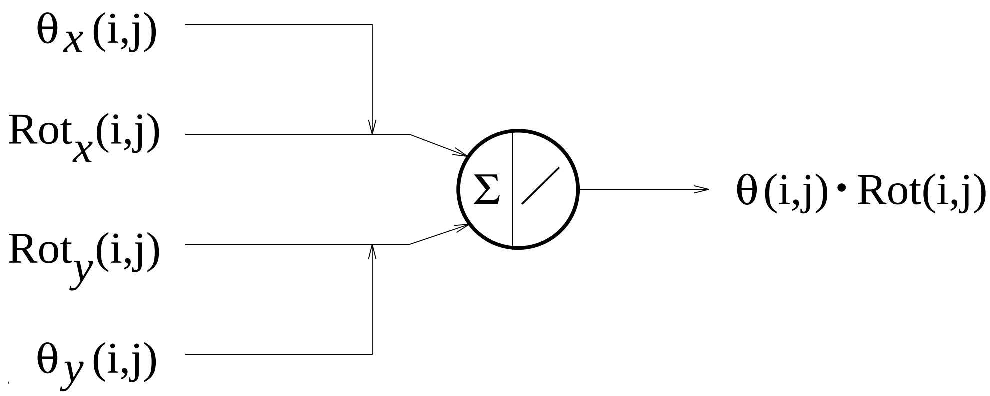



Figure 6.6: Calculation of the dot product of the tangent estimate $\theta$ and the vector field corresponding to rotation, for the image point at coordinates $(i, j)$.

As an example, consider the subsystem corresponding to the point in the input image at coordinates $(i, j)$. Figure 6.6 shows the neural implementation of the first stage of the calculation. Modules of this type are replicated for each point in the input image, and for each Lie vector field image. All that then remains is to pass the output of these modules through modules which calculate the absolute value of their input.

Figure 6.7 shows a neural module which calculates a good approximation of the absolute value of its input. It is less accurate for inputs very close to zero, but in this application, where possible orientations are coarsely quantized, this does not cause any problems. This completes the neural calculation of the Local Measure of Consistency for each contour point with respect to each Lie vector field. All that remains is to combine these into an Invariance Signature.

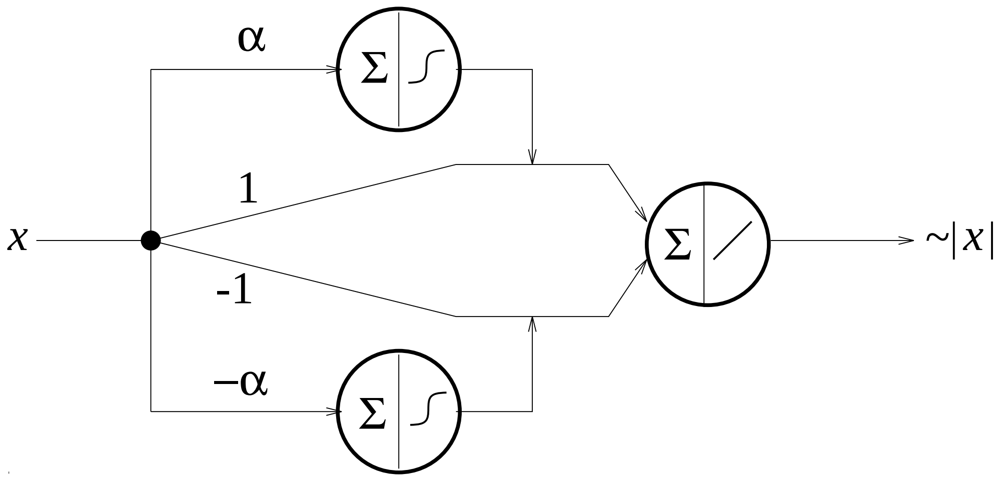



Figure 6.7: Neural module which calculates the absolute value of its input.

## 6.3 Calculation of the Invariance Signature

### 6.3.1 The Binning Unit

The ability of the MBNN approach to produce modules that perform tasks not usually associated with neural networks is illustrated by the binning unit shown in Figure 6.8. There is one of these modules for each of the $n$ bins comprising the Invariance Signature histogram for each invariance class. Each binning module is connected to all the nodes in the Invariance Image, and inputs to it are gated by the binary input image, so that only the $N$ nodes corresponding to ones in the input image contribute.

The $n$ bins have width $\frac{1}{n-1}$, since the first bin is centred on 0, and the last on 1.[6] The system is designed so that Nodes A and B only have an output of 1 when the input $x$ is within bin $i$. The condition for $x$ to be in bin $i$ is:

$$\frac{2i-1}{2(n-1)} < x < \frac{2i+1}{2(n-1)} \tag{6.8}$$

[6]This is because a bin ending at an extreme of the range would miss contributions from the extreme value since the edge of the neural bin is not vertical.

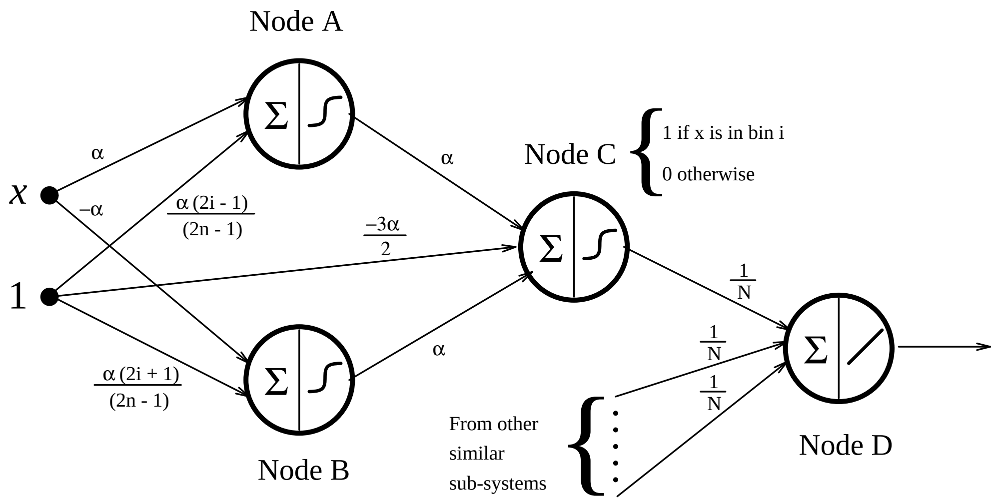



Figure 6.8: Neural binning module.

In order to detect this condition, the activations of Nodes A and B are set to:

$$net_A = \alpha x - \frac{\alpha(2i-1)}{2(n-1)} \tag{6.9}$$

$$net_B = -\alpha x + \frac{\alpha(2i+1)}{2(n-1)}. \tag{6.10}$$

The outputs of these nodes go to Node C, which computes a neural AND. Node D sums the contributions to bin $i$ from all N nodes.

This concludes the calculation of the Invariance Signature. The end result is a signature which consists of $3n$ values, where $n$ is the number of bins in the invariance signature histogram for each invariance class. This calculation has been achieved in the framework of a modular MBNN, where module functions are strictly defined, and are completely independent of any training data. The building blocks of the individual modules are restricted to simple summation artificial neurons, with either linear or sigmoidal transfer functions. The sole departure from standard neural network components is the introduction of dynamic weights, but, as already stated, these can be eliminated if product neurons are used, as in Higher Order Neural Networks (see §3.4.3). This shows the power of the modular approach to neural network design, and demonstrates that it is possible to design modules which perform tasks that are not often considered to be part of the neural networks domain, such as binning data.

A final module can be added to this network which is trained to classify patterns on the basis of the $3n$-dimensional Invariance Signature, rather than on the input patterns themselves. Classification performance will thus be guaranteed to be invariant under shift, rotation and scaling. The advantages of this approach are demonstrated in Chapter 7.