CHAPTER 5

NAVIGATION

This chapter presents a connectionist control acquisition architecture tailored for simulated short-range homing in the presence of obstacles. The kinematics of a cylindrical robot (Figure 5.1) moving in a planar environment are simulated. The robot has wheels that propel it independently and simultaneously in both the x and y directions with respect to a fixed orientation. It can move up to one radius per discrete time step. Each obstacle in the robot's environment has an associated real-valued grey level between 0 and 1 representing the darkness of the grey; the higher the value, the darker the grey, with 0 corresponding to white and 1 corresponding to black. The robot has a 360 degree sensor belt with 64 distance sensors and 64 grey-scale sensors evenly placed around its perimeter. These 128 values are smoothed in a preprocessing step to produce 32 values that form the robot's *view*.

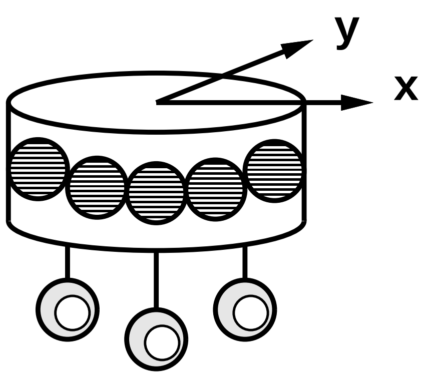

Figure 5.1. The Simulated Robot.

Figure 5.2 is a display created by the navigation simulator. The bottom portion of the figure shows a bird's-eye view of the robot's environment. In this display, the bold circle represents the robot's "home" position, with the radius line indicating the fixed home orientation. The other circle with radius line represents the robot's current position and fixed orientation. The top panel shows the grey-scale view from the home position, and the next panel down shows the grey-scale view from the robot's current position. For the reader's better viewing, the distance and grey-scale sensor values are shown on the same panel, where the height of the profile is the reciprocal of the distance. Thus as the robot gets closer to objects they get taller in this display, and when the robot gets farther away from objects they get shorter.

The robot can neither move nor "see" through obstacles (i.e., obstacles are opaque). The task is for the robot to align itself with the home position from arbitrary starting positions in the environment while not colliding with obstacles. This task is performed using only the sensory information—the robot does not have access to the bird's-eye view.

This is a difficult control task. Although the sensory information provides complete state information, it forms a high-dimensional continuous task, and successful homing generally requires a nonlinear mapping from this space to the space of real-valued actions. Further, it is not easy to train networks on this space. The robot has no comprehensive world model and receives no global information; it receives only sensory information that is inherently limited. Furthermore, because it is difficult to reach home using random exploration due to the large number of environmental states, simple trial-and-error learning is intractable. In order to solve this task an architecture was designed that facilitates the coding of generic domain knowledge in the form of hard-wired networks, architectural constraints, and initial weights.

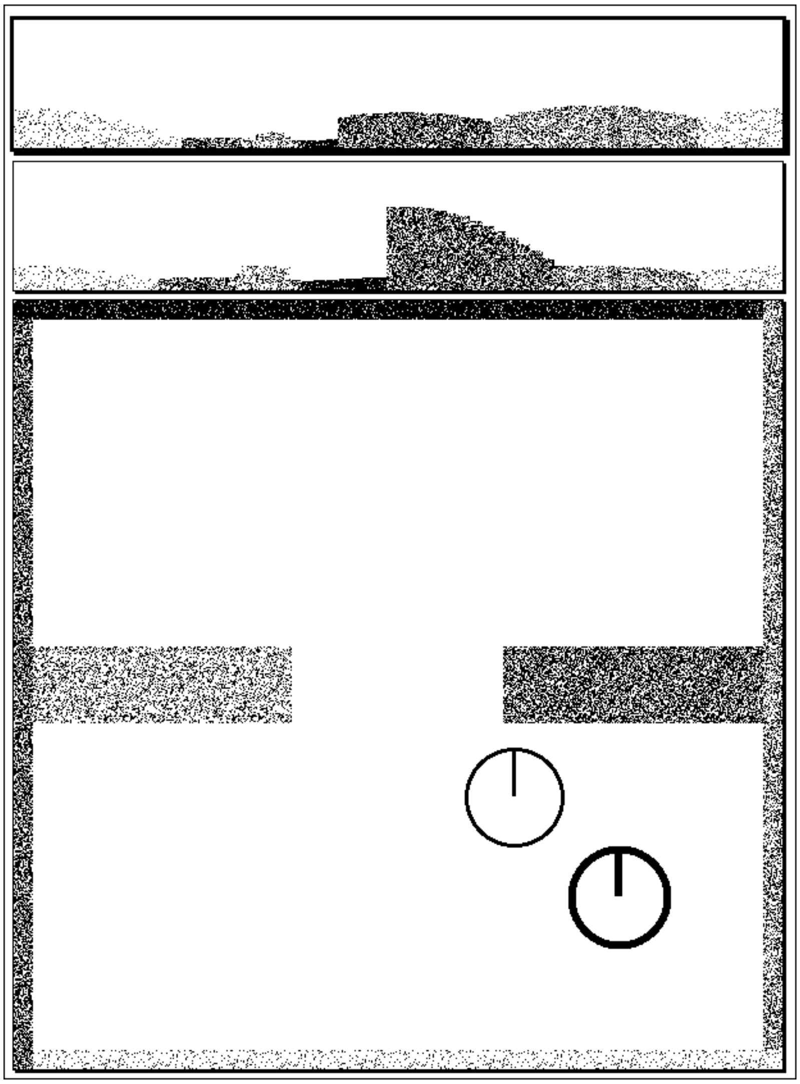

Figure 5.2. The Navigation Simulator.

5.1 Artificial Potential Functions

Before we describe our architecture, we briefly discuss a more traditional technique for navigation that uses artificial potential functions (e.g., Connolly, Burns, and Weiss [15], Khatib [40], Koditschek [43], Krogh [45], LaTombe [47], and Koditschek and Rimon [71]). This technique involves building explicit object models representing the extent and position of objects in the robot's environment. Repelling potential fields are then placed around obstacles using the object models, and the center of an attracting potential field is placed on the goal. This can be visualized as a terrain where the global minimum is located at the goal, and where there are bumps around the obstacles. The height at each point in this terrain is the sum of all the potential functions and will be called the *navigation function* [71]. The robot goes home by descending the terrain.

The contour diagram in Figure 5.3 shows such a terrain. The task is to go from the top room to the home position in the bottom room via the door. Unfortunately, there can be local minima. [1] In this environment there are two prime examples of minima: the right-hand wall between the home location and the upper room, where opposing forces exactly counteract each other to produce a local minimum in the right-hand side of the upper room, and the doorway, where the repelling fields on the door frame create an insurmountable bump in the center of the door as well as a local minimum just above the doorway.

In contrast this traditional technique, our technique involves creating a sensor-based potential function model through learning. Instead of hard-wiring static potential functions from the object models, the proposed architecture learns potential functions, automatically adjusting them to both avoid local minima and to produce

[1] Rimon and Koditschek [71] present a technique that produces a navigation function with a single minimum for a restricted class of environments. Connolly, Burns, and Weiss [15] present a technique for computing a navigation function with a single local minimum. This navigation function can be used for planning smooth robot paths and their technique works for all environments. Like the other artificial potential function techniques, both of these techniques assume explicit object models.

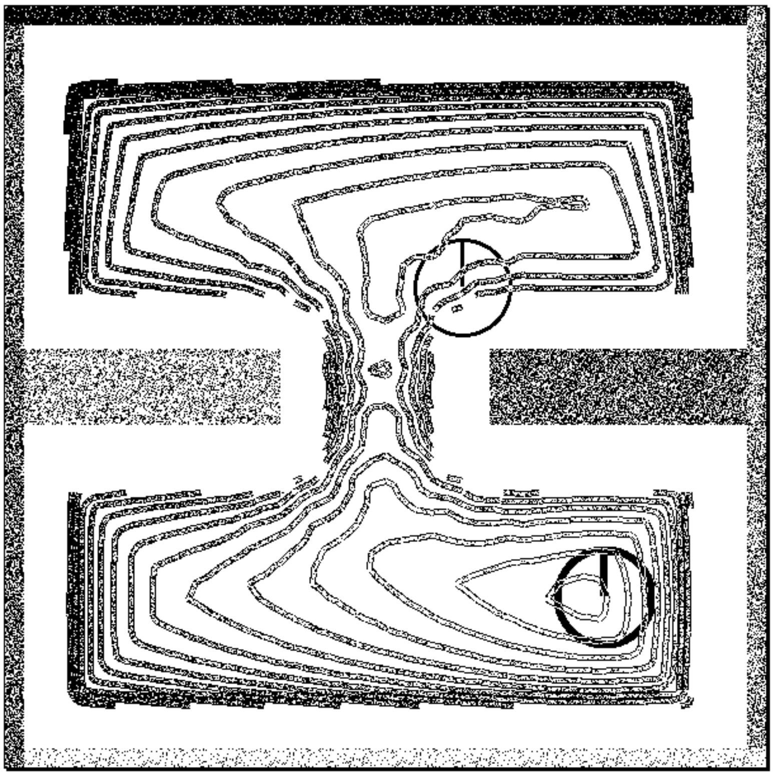

Figure 5.3. The Artificial Potential Function Method. This figure shows a contour plot of a terrain created using artificial potential functions generated from object models. The contour diagram shows level curves where the grey level of the line depicts the height of the line: the maximum height is depicted in black, and the minimum height is depicted in white.

efficient trajectories. Furthermore, it does this without object models, using only sensory information.

5.2 The Navigation Architecture

The navigation architecture is a version of our control acquisition architecture (described in Chapter 3.7) tailored for navigation, where the state is the robot's view and the evaluation is an estimate of the length of the shortest path from the robot's current location to home. The basic architecture is shown in Figure 5.4 and will be called the *navigation network*. It consists of a *range-flow model* and two adaptive critics. The range-flow model maps view/action pairs to next views; the *homing critic* maps views to path length home using a straight line trajectory; the *obstacle-avoidance critic* maps views to estimated minimum additional path length needed to avoid obstacles. The sum of the outputs of the homing critic and the obstacle-avoidance critic equals the total estimated shortest path length home. The range-flow model is a hard-wired differentiable network incorporating geometric knowledge about the sensors and space. Each critic is a radial basis network using Gaussian hidden units.

The controller is implicit in this architecture. At each time step the action is chosen that locally minimizes the sum of the outputs of the critics. Because the actions are real-valued, it is intractable to present all possible actions in turn as input to the navigation network. Instead, gradient descent is used to find a feasible action $(\Delta x, \Delta y)$ that minimizes the output of the sum of the current outputs of the critics, where feasible means that the action has length less than the radius of the robot. This action—the one that is the first action in a sequence of actions that yields the minimum path length needed to reach the goal state according to the current critics' estimates—is then executed.

Before describing the range-flow model and the critics, we first describe the robot's view in more detail. The view is constructed using 64 distance sensors

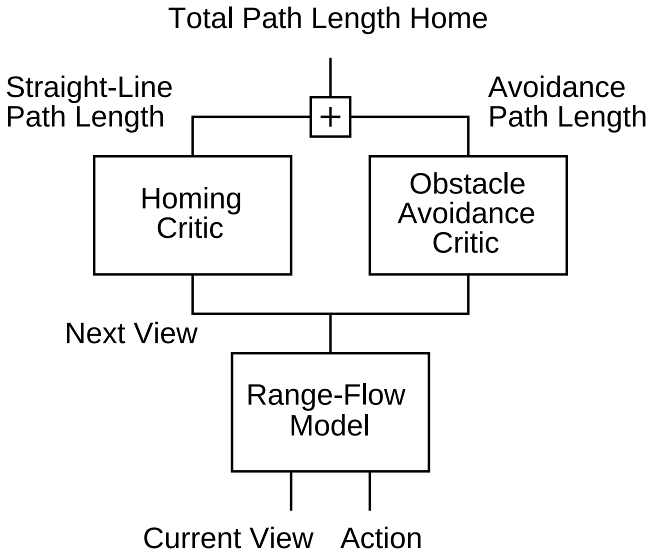

Figure 5.4. The Navigation Architecture.

and 64 grey-scale sensors evenly placed around the robot's perimeter. The values on these 128 sensors are smoothed in a preprocessing step to create 32 values that comprise the robot's view. We denote the 64 distance measurements and the 64 grey-scale measurements by $(\breve{\boldsymbol{d}}, \breve{\boldsymbol{g}}, \breve{\boldsymbol{\theta}})$, where $\breve{\boldsymbol{d}}$ is the vector of distance measurements, $\breve{\boldsymbol{g}}$ is the vector of grey-scale measurements, and $\breve{\boldsymbol{\theta}}$ is the vector of fixed sensor angles, with $\breve{\theta}_k = \frac{2k\pi}{64}$, $k = 0, \ldots, 63$. Each distance measurement $\breve{d}_k$ is the distance to the closest obstacle in the direction of its respective sensor angle $\breve{\theta}_k$, and each grey-scale measurement $\breve{g}_k$ is the grey-value of the closest obstacle in the direction of its respective sensor angle $\breve{\theta}_k$. We denote the 32 smoothed values by $(\boldsymbol{d}, \boldsymbol{g}, \boldsymbol{\theta})$, where $\boldsymbol{d}$ is a vector of smoothed distance values, $\boldsymbol{g}$ is a vector of smoothed distance values, and $\boldsymbol{\theta}$ is the vector of angles corresponding to these smoothed values, where $\theta_k = \frac{2k\pi}{16}$, $k = 0, \ldots, 15$.

The sensor measurements are smoothed with a bank of 16 Gaussian smoothing units for each sensory modality (i.e., distance and grey-scale). The j^{th} Gaussian smoothing unit has a receptive field represented by a Gaussian distribution with mean θ_j and fixed variance σ. The variance is such that the Gaussian distribution has value .5 at the midpoint between two neighboring angles θ_k and θ_{k+1}. This produces overlapping receptive fields and gives good smoothing performance in practice. The j^{th} Gaussian distance smoothing unit weights each distance measurement $\breve{d}_i$ by the extent to which the corresponding angle $\breve{\theta}_i$ falls in its receptive field, where the extent is calculated with a Gaussian function as shown below. Finally, this Gaussian distance smoothing unit computes its output by summing these weighted measurements and then normalizing by the total amount of weight contributed by these measurements:

$$d_j = \frac{\sum_i \breve{d}_i e^{-(\theta_j - \breve{\theta}_i)^2/\sigma}}{\sum_i e^{-(\theta_j - \breve{\theta}_i)^2/\sigma}}. \tag{5.1}$$

Grey-scale smoothing proceeds in exactly the same fashion. Figure 5.5 shows this smoothing process for the case of 16 distance sensors and 4 Gaussian distance smoothing units.

For the purpose of this description, it is easiest to think of these 32 smoothed values as corresponding to 32 virtual sensors that have overlapping Gaussian receptive fields. In order to simplify the description, throughout the rest of the paper we treat these virtual sensors as the robot's real sensors.

The hard-wired range-flow model predicts the next view given an action and the current view. The range-flow model forms its prediction in two steps: change of coordinates followed by interpolation. The distance values of the current view and their respective angles $(\boldsymbol{d}, \boldsymbol{\theta})$ define a set of points specified in polar coordinates on obstacles in the robot's environment relative to the robot's own body-centered coordinate system. This is shown in Figure 5.6. When the robot moves, it can recalculate these points $(\boldsymbol{d}, \boldsymbol{\theta})$ with respect to its new coordinate system giving $(\boldsymbol{d}', \boldsymbol{\theta}')$. This step is simply a change of coordinates. But to predict the sensor values, the robot needs to predict the distances corresponding to the sensors' fixed angles $\boldsymbol{\theta}$ instead of $\boldsymbol{\theta}'$. To do this, it interpolates the $(\boldsymbol{d}', \boldsymbol{\theta}')$ points to form distance predictions aligned with the original angles $\boldsymbol{\theta}$, resulting in $(\hat{\boldsymbol{d}}, \boldsymbol{\theta})$. This process is shown in Figure 5.6. The

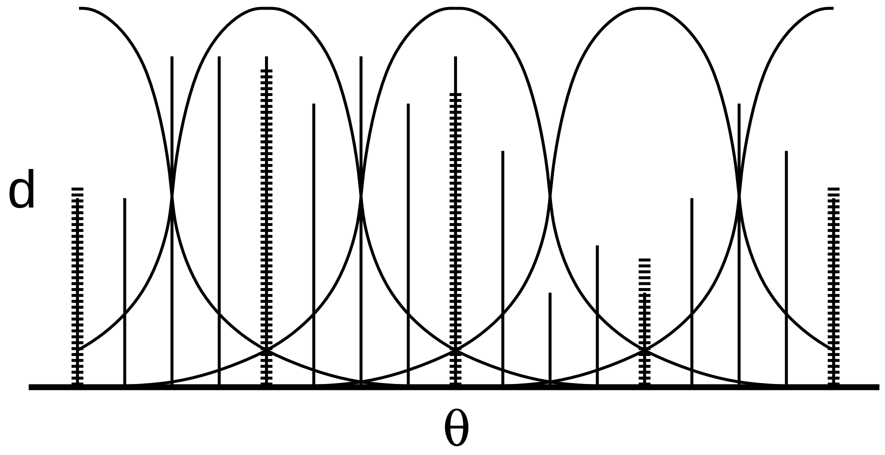

Figure 5.5. The Sensor Value Smoothing Process. It is shown for the case of 16 distance measurements that are used to produce 4 smoothed values. All sensor values are shown in polar coordinates using a bar-chart plotting style, where the top of each line defines its polar coordinates. The sensor measurements are represented as solid lines, and the smoothed values are represented as dashed lines. Because of wrap-around, the leftmost and rightmost sensor values are the same.

$\boldsymbol{\theta}'$ sensor angles are also used to interpolate the grey-scale values $\boldsymbol{g}$ to form grey-scale predictions $\hat{\boldsymbol{g}}$.

It is important to distinguish our range-flow model from a model of the robot's environment and kinematics. A range-flow model merely predicts how the observable range data (i.e., points measured on obstacles in the robot's environment) will be transformed upon the execution of a proposed action. In this fashion, it is independent of any particular environment, and therefore does not have to be retrained on new environments. Unfortunately, this generality has its price: the range-flow model is inexact. There are two major sources of errors in a range-flow model: approximation error due to interpolation, and prediction error due to environmental influences. In contrast to an environment-specific model, the range-flow model cannot handle certain environmental influences. In particular, the range-flow model cannot predict the effects of infeasible actions or the emergence into view of hidden objects; in general, it cannot cope with discontinuities in the environment. If we assume that environ-

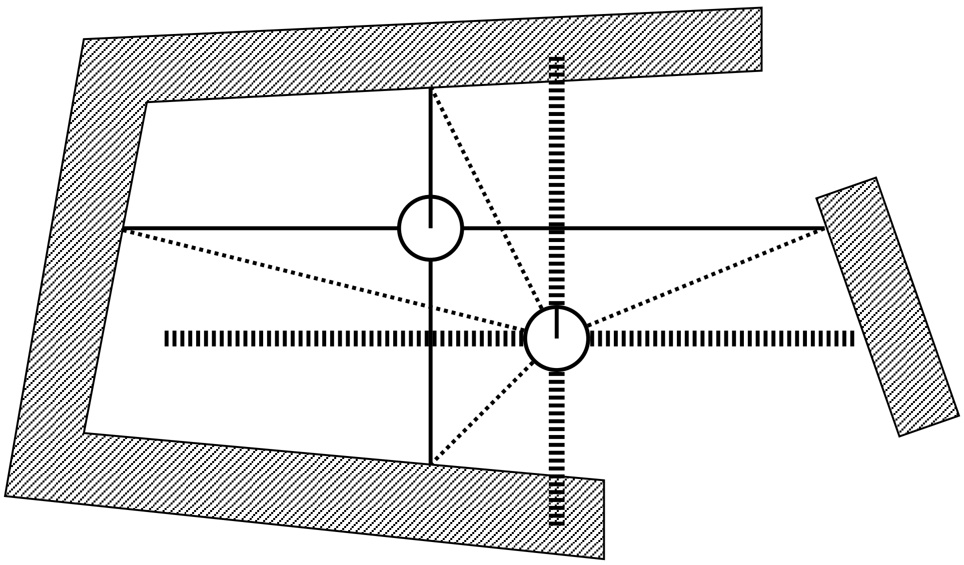

Figure 5.6. The Operation Of The Range-Flow Model. The distance values of the robot's view are shown as a set of vectors to points on objects in the robot's environment. The left robot represents the robot before a translation, and the right robot represents the robot after a translation. The robot has four evenly spaced distance sensors. (Note that the actual robot has more sensors.) The solid rays emanating from the left robot's center represent vectors to the closest points on obstacles along the sensors' angles. The narrow dashed rays emanating from the right robot's center represent vectors to those same points but with respect to the robot's new body-centered coordinate system. The thick dashed rays emanating from the right robot's center show the interpolated distance predictions along the sensors' angles. Although there will always be prediction errors, the ones depicted in this example are much larger than would be expected in practice, where there would be many more sensors, and the actions would be much smaller.

ments are continuous almost everywhere, then we can bound the approximation and prediction errors. In particular, the range-flow model's accuracy increases with an increased number of sensors, and its accuracy decreases with an increase in the size of the robot's movement.

The connectionist range-flow model consists of a set of hard-wired units that perform the translation and interpolation as shown in Figure 5.7. [2] The translation units simultaneously transform the input view from polar to rectangular coordinates and calculate the translation. In particular, the x translation units compute their outputs as follows:

$$x'_j = d_j \cos\theta_j - \Delta x, \tag{5.2}$$

where d_j is the j^{th} distance measurement, θ_j is the sensor angle corresponding to d_j, and Δx is the x translation. The y translation units compute their outputs as follows:

$$y'_j = d_j \sin\theta_j - \Delta y, \tag{5.3}$$

where Δy is the y translation. The rectangular-to-polar units compute their outputs as follows:

$$d'_j = \sqrt{x'^2_j + y'^2_j}, \tag{5.4}$$

and

$$\theta'_j = \tan^{-1}(x'_j, y'_j). \tag{5.5}$$

Interpolation calculations are performed by a series of Gaussian interpolation units, one per sensor. Gaussian interpolation was chosen because it is simple and differentiable. The j^{th} Gaussian interpolation unit has a receptive field, by a Gaussian distribution with mean θ_i and fixed variance σ. The variance is set such that the Gaussian distribution has height .5 at the midpoint between two neighboring

[2] Although the transformation could be calculated without the use of a network, in using a network, the model can be easily integrated into the entire navigation network. Furthermore, back-propagation can be employed in order to efficiently compute derivatives.

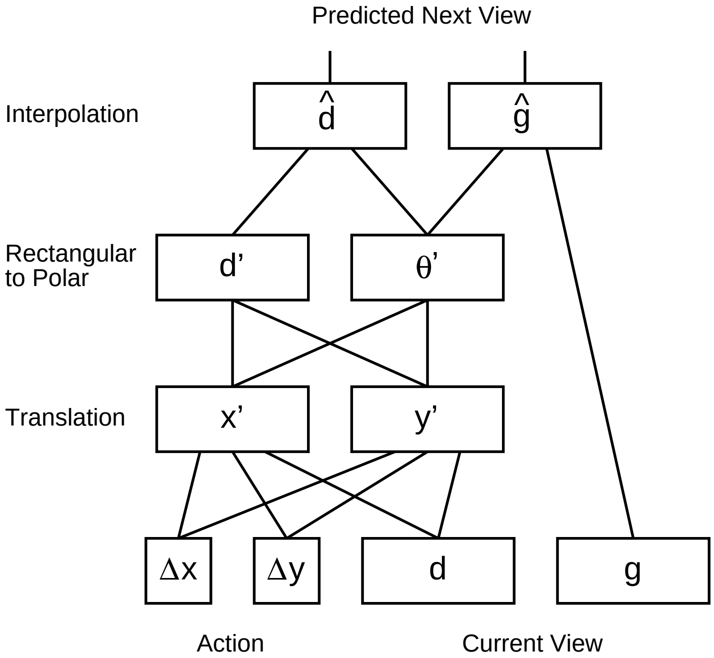

Figure 5.7. The Range-Flow Model Network. In this figure, $\boldsymbol{d}$ is a vector of distance measurements, $\boldsymbol{g}$ is a vector of grey-scale measurements, $(\boldsymbol{x}', \boldsymbol{y}')$ are the translated points in cartesian coordinates, $(\boldsymbol{d}', \boldsymbol{\theta}')$ are the translated points in polar coordinates, and $\hat{\boldsymbol{d}}$ and $\hat{\boldsymbol{g}}$ are respectively the interpolated distance and grey-scale values aligned with the sensor angles $\boldsymbol{\theta}$.

sensor angles θ_k and θ_{k+1}. This produces overlapping receptive fields and gives good interpolation performance in practice.

The range-flow model calculates the points $(\boldsymbol{d}', \boldsymbol{\theta}')$ on objects in the robot's environment with respect to the robot's body-centered coordinate system after a proposed translation. The j^{th} Gaussian interpolation unit takes each of these points (d'_i, θ'_i) and weights each distance value d'_i by the extent to which the angle θ'_i falls in its receptive field, where the extent is calculated with a Gaussian function as shown below. Finally, a Gaussian interpolation unit computes its output by summing these weighted distances and then normalizing by the total amount of weight contributed by these points:

$$\hat{d}_j = \frac{\sum_i d'_i e^{-(\theta_j - \theta'_i)^2/\sigma}}{\sum_i e^{-(\theta_j - \theta'_i)^2/\sigma}}. \tag{5.6}$$

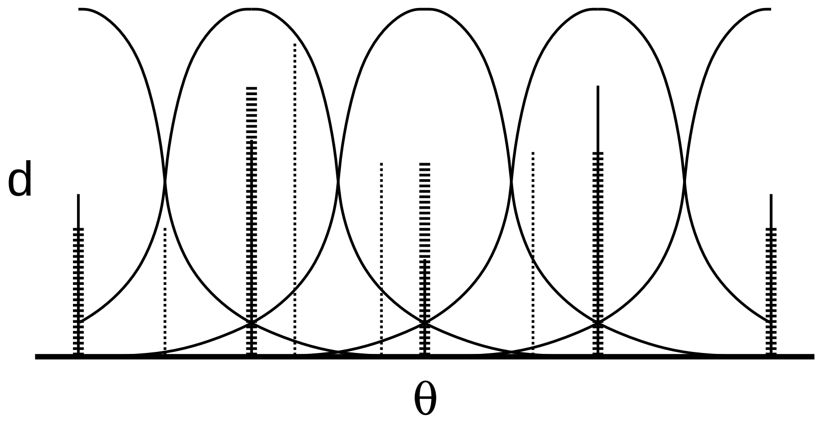

Figure 5.8. Gaussian Interpolation. The rays from Figure 5.6 are shown in polar coordinates, with angle on the abscissa and distance on the ordinate. The tops of the lines represents the end points of the vectors. The robot has four evenly spaced distance sensors. The solid lines represent the closest points on obstacles along the sensors' angles. The narrow lines represent those same points after a change of coordinates. The thick dashed lines represent the interpolated distance predictions along the sensors' angles. In this case, four Gaussians interpolation units perform the interpolation step. These units are represented by four Gaussian distributions evenly spaced across the abscissa. Because of wrap-around, the leftmost prediction is the same as the rightmost. The leftmost Gaussian distribution shows the case of a single unaligned point (where the other points' contribution is negligible). In this case, the interpolation scheme aligns the point's angle with the sensor's angle while maintaining the original distance. The second Gaussian distribution from the left shows the case of multiple unaligned points. In this case, Gaussian interpolation produces a distance prediction that is a weighted average of all the points.

This calculation handles both the case of unaligned angles and the case of multiple θ_i''s mapping to a particular θ_j.

The basic operation of these units is shown in Figure 5.8. The Gaussian interpolation process is very similar to the Gaussian smoothing process. The difference between the two processes is that smoothing reduces the dimensionality of the input and has fixed input angles, whereas the interpolation process maintains the dimensionality of the input and has variable input angles. These similarities and differences can be seen by comparing Figures 5.5 and 5.8.

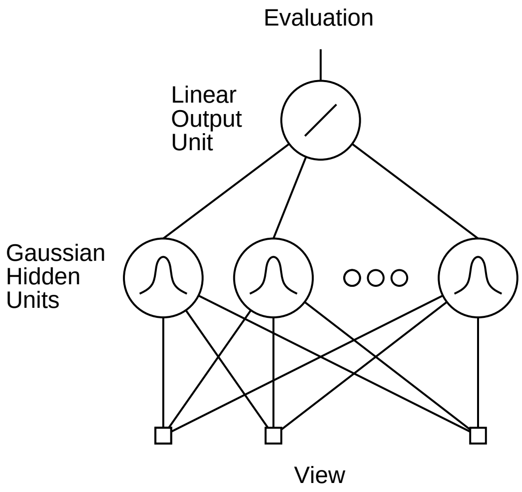

Figure 5.9. A Critic Network.

Each critic network is composed of two layers of units as shown in Figure 5.9. The output layer consists of a single linear unit. The hidden layer consists of a number of Gaussian units which contain vector mean and scalar variance parameters. These units have Gaussian receptive fields and calculate their activities based on the responses of their receptive fields to their inputs:

$$y_j = e^{\frac{\sum_i (x_{ij} - w_{ij})^2}{\sigma_j}}, \tag{5.7}$$

where y_j is the output of the j^{th} Gaussian unit, x_{ij} is the i^{th} input to the j^{th} Gaussian unit, w_{ij} is the i^{th} weight to the j^{th} Gaussian unit, and σ_j is the variance of the j^{th} Gaussian unit.

5.3 Training

The training process is divided into three major phases. Initially the critics' Gaussian hidden units are trained using an unsupervised learning procedure in order to produce good initial weights that place the Gaussian distributions in regions of view space that are densely populated by views from the particular environment. Next, the homing critic is trained, and finally, while the weights of the homing critic are fixed, the avoidance critic is trained. Throughout the entire training process the range-flow model remains fixed.

The critic networks' Gaussian hidden units are trained initially using a maximum likelihood technique called *expectation maximization*, which is a soft form of competitive learning (see Duda and Hart [16] and Nowlan [63, 62] for details). The units are trained on a number of views generated by the robot from a uniform distribution of positions in the environment. Expectation maximization assumes a parameterized class of models, in this case an additive mixture of multidimensional spherical Gaussian distributions, and it selects the model from that class that is most likely to have produced the data, in this case the views. In short, this process produces a set of Gaussian units that best cover the view data. Figure 5.10 shows the means and variances of the Gaussian distributions after training.

The Gaussian units turn out to be spatially-tuned, that is, they are most responsive to a spatially local region of the robot's workspace, despite the fact that they are trained only on view data. Figure 5.11 shows the response of a few Gaussian units to views generated from a grid of evenly spaced positions in the door environment shown in Figure 5.2. Each Gaussian unit covers a spatially local region of the robot's workspace. Some reasons this occurs are 1) the view space is smooth, 2) by inspection, there appears to be a one-to-one mapping between views and positions in the robot environments that we consider, and 3) the Gaussian units have local receptive fields. Figure 5.12 shows the sum of the responses of all the Gaussian units to this grid of views and demonstrates that the Gaussian distributions provide good coverage of the robot's workspace. In summary, each position in the robot's workspace activates at least one, and no more than a few, Gaussian units. The set of weights produced by this procedure provides a good starting point for training the critics.

Initially, the navigation network is trained to produce straight-line trajectories to home. The homing critic is trained using *dead-reckoning*. Dead-reckoning is a technique for keeping track of the distance to home by accumulating the incremental

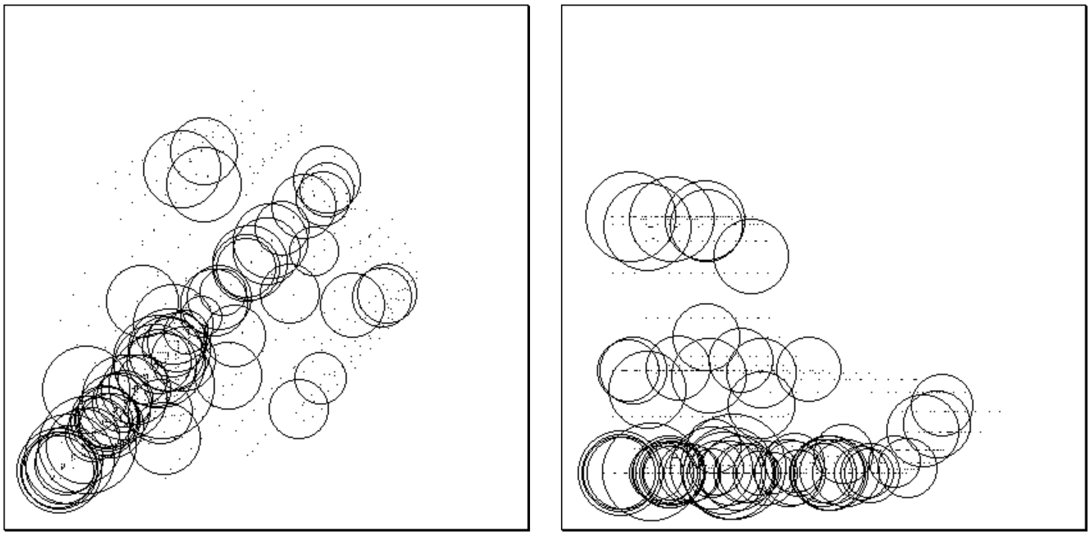

Figure 5.10. The Clustering Phase. The two panels show the Gaussian distributions' coverage of view space using 64 Gaussian distributions plotted on two different two dimensional slices of view space. The view data are depicted as points that were generated at each of the feasible positions of a uniform 50×50 grid that was superimposed on the robot's environment, where a feasible position is a location where the robot does not intersect an obstacle. Each Gaussian distribution is indicated by a circle whose center position is the mean and whose radius is the variance. The left panel plots the views and Gaussian distributions using the values of the first two distance sensors (i.e., θ_0 and θ_1) as the abscissa and ordinate respectively. The right panel plots the views and Gaussian distributions using the values on the first two grey-scale sensors as the abscissa and ordinate respectively.

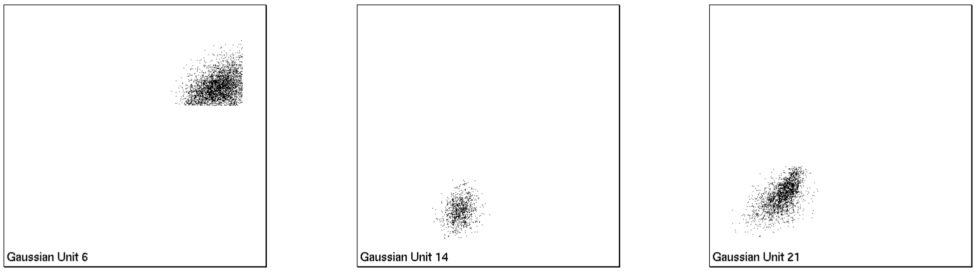

Figure 5.11. The Response of Single Gaussian Hidden Units Across the Robot's Environment. The three panels show the response of single Gaussian units to views generated at each of the feasible positions of a uniform 50×50 grid that was superimposed on the robot's environment (shown in Figure 5.3), where a feasible position is a location where the robot does not intersect an obstacle. Furthermore, the responses of all 64 Gaussian units at each spatial position sum to one. The real-valued responses are plotted as grey-values, where the value one is depicted as black, and the value zero is depicted as white.

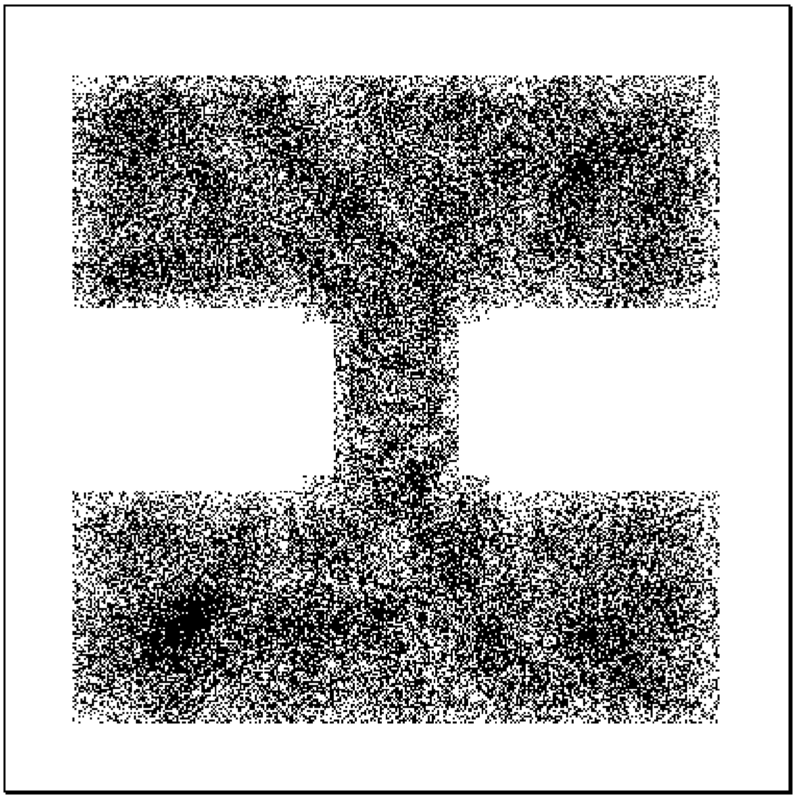

Figure 5.12. The Total Response of All the Gaussian Hidden Units Across the Robot's Environment. Shown are the sums of the responses of all 64 Gaussian units to views generated at each of the positions of a uniform 50×50 grid that was superimposed on the robot's environment (shown in Figure 5.3). To show the actual coverage across the robot's workspace, the responses of all the Gaussian units at each spatial position are not constrained to sum to one. The real-valued responses are plotted as grey-values, where the highest response is black and the lowest response is white.

displacements during exploration from home. This distance provides the target output for training the homing critic via supervised learning. While the robot randomly explores the environment, the weights of the homing critic—the weights of both the output unit and the hidden layer—are adjusted using back-propagation. After training, the output of the critic on a particular view will be the straight-line distance to the home position from the position in the robot's environment that generated that view, and a 3-d surface plot of the output of the homing critic versus position is a cone having a minimum at the home position and increasing with constant slope with increasing distance from the home position.

Next, the navigation network is further trained to avoid obstacles. In this phase, the obstacle-avoidance critic is trained while the weights of the homing critic are frozen. Using a temporal difference (TD) method (Sutton [92, 93] and Barto, Sutton, and Anderson [8]), the obstacle-avoidance critic is adjusted so that the expected path length decreases by one radius per-time-step. After training, the robot takes successive one-radius steps toward its home location.

The training during this phase is divided into a series of trials, each of which begins in the home position and ends when the robot either is inside one radius of the home position or collides with an obstacle. At the beginning of each trial, the robot wanders for a fixed number of steps by taking actions selected randomly from a uniform distribution. During this random walk the robot avoids collisions with obstacles by only executing feasible actions. If an action results in a collision then the action's length is clipped so that the robot stops just short of the obstacle. This random walk places the robot in a random initial state. During the rest of the trial, the robot uses both critics to choose actions resulting in minimal predictions of the path length to reach the home position. Gradient descent is used to choose the best action. The details of the action selection procedure are summarized in Table 5.1.

After each move the obstacle-avoidance critic is trained using a TD method. A system using this method learns to predict by maintaining consistency across

Table 5.1. The Gradient Descent Procedure for Choosing Actions. r is the radius of the robot, η is a learning rate constant, and ϵ is a small constant.

1. Initialize $(\Delta x, \Delta y) = (0, 0)$.
2. Loop
   (a) Feed $(\Delta x, \Delta y)$ and the current view into the network and calculate the predicted total path length to the home position $p(t)$.
   (b) Compute $(-\frac{\partial p(t)}{\partial \Delta x}, -\frac{\partial p(t)}{\partial \Delta y})$ using back-propagation. (This can be computed by back-propagating a -1 into the navigation network.)
   (c) $(\Delta x, \Delta y) \leftarrow (\Delta x - \eta \frac{\partial p(t)}{\partial \Delta x}, \Delta y - \eta \frac{\partial p(t)}{\partial \Delta y})$.
3. Until $|(-\frac{\partial p(t)}{\partial \Delta x}, -\frac{\partial p(t)}{\partial \Delta y})| < \epsilon$ or $|(\Delta x, \Delta y)| > r$.
4. If $|(\Delta x, \Delta y)| > r$ then $(\Delta x, \Delta y) \leftarrow (\frac{r \Delta x}{|(\Delta x, \Delta y)|}, \frac{r \Delta y}{|(\Delta x, \Delta y)|})$. This step ensures the feasibility of the movement.

successive predictions and by forcing the prediction to agree with any available training signals. In the case of predicting the path length to the home position, the critic is trained to make successive predictions decrease by one radius and to make the prediction at the home position zero. Furthermore, the critic is penalized when the robot hits an obstacle. The exact error signal is:

$$e(t) = \begin{cases} h, & \text{if the robot just hit an obstacle;} \\ d(t) - p(t), & \text{if the robot is within one radius of the home position;} \\ p(t+1) - p(t) - r, & \text{otherwise,} \end{cases} \tag{5.8}$$

where h is a fixed penalty factor greater than zero for hitting an obstacle, $d(t)$ is the distance from the center of the robot to the center of the goal position at time t, $p(t)$ is the sum of the outputs of the homing and obstacle-avoidance critics at time t, and r is the radius of the robot. This error signal is back-propagated to adjust the obstacle-avoidance critic's weights.

Table 5.2. The Gradient Descent Procedure for Perceptual Servoing. $(\Delta x, \Delta y)$ is an action, $d(t)$ is the distance between the position of the center of the robot and the home position, η is a learning rate, and δ and ϵ are small constants.

1. Initialize $(\Delta x, \Delta y) = (0, 0)$.
2. While $d(t) > \delta$ Do Begin
   (a) Loop
      i. Feed $(\Delta x, \Delta y)$ and current view into the range-flow model and calculate predicted view.
      ii. Back-propagate the difference between the prediction and the home view giving $(-\frac{\partial E}{\partial \Delta x}, -\frac{\partial E}{\partial \Delta y})$.
      iii. $(\Delta x, \Delta y) \leftarrow (\Delta x - \eta\frac{\partial E}{\partial \Delta x}, \Delta y - \eta\frac{\partial E}{\partial \Delta y})$.
   (b) Until $|(\frac{\partial E}{\partial \Delta x}, \frac{\partial E}{\partial \Delta y})| < \epsilon$
   (c) Execute $(\Delta x, \Delta y)$.
3. End

5.4 Perceptual Servoing

Our differentiable range-flow model can be used to perform *perceptual servoing*, a technique for determining an action given only perceptual information (see Fennema, Hanson, and Riseman [23]). It calculates a single action for getting home given only the current view and the home view. This is useful for final homing when the robot is one action away from the home location. Our technique uses gradient descent to choose an action such that the predicted next view is the view generated from the home position. The procedure is summarized in Table 5.2. Figure 5.13 shows two example trajectories created using perceptual servoing in the door environment.

Perceptual servoing can be used in conjunction with the navigation technique described earlier to form complete trajectories. Initially actions are chosen that minimize the predicted path length to the home position. Then perceptual servoing is employed when the robot is within one radius of the home position.

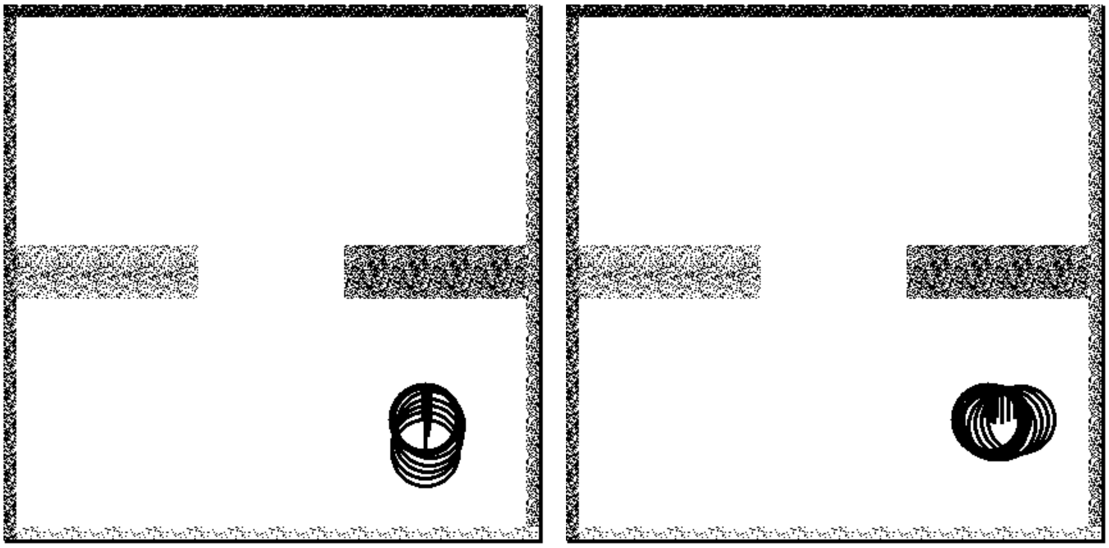

Figure 5.13. Two Examples of Perceptual Servoing in the Door Environment.

5.5 Results

The architecture was applied to the obstacle environment shown in Figure 5.14 and the door environment shown in Figure 5.2. The homing and obstacle-avoidance critic networks were composed of 200 Gaussian units, whose weights were trained initially using expectation maximization on views generated from the feasible positions of an evenly spaced grid of 64×64 positions superimposed on the robot's environment. Expectation maximization converged in approximately 200 iterations. These weights were then used as the initial weights for the hidden layer of both critics. The weights of each critic's output unit were initialized to zero. A learning rate of 0.025 was used for training the homing critic and 0.0005 for training the obstacle-avoidance critic. The penalty factor h was chosen to be 12 in all the experiments.

Figures 5.15 and 5.16 show the results of training. The left panel of each figure is a contour plot of the output of the homing critic and reflects only the straight-line distances to the home location from each respective starting position. The right panel of each figure is a contour plot of the combined outputs of the homing critic and the obstacle-avoidance critic. This plot now reflects the actual path length home.

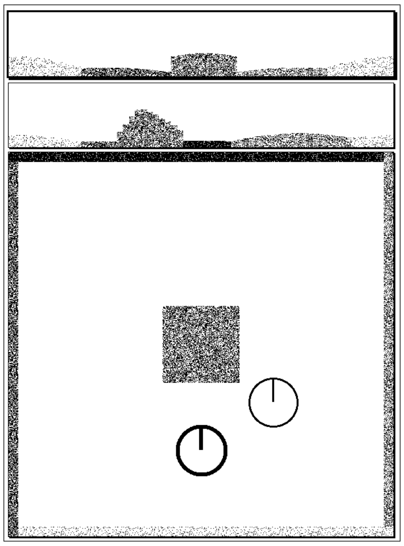

Figure 5.14. The Obstacle Environment.

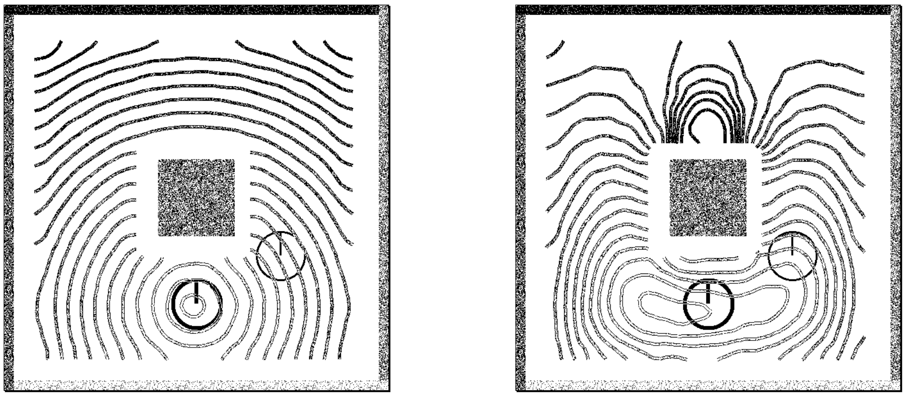

Figure 5.15. Contour Plots of the Output of the Critics After Training on the Obstacle Environment. The left panel shows a contour plot of the output of the homing critic. The right panel shows a contour plot of the sum the outputs of the homing and obstacle-avoidance critics.

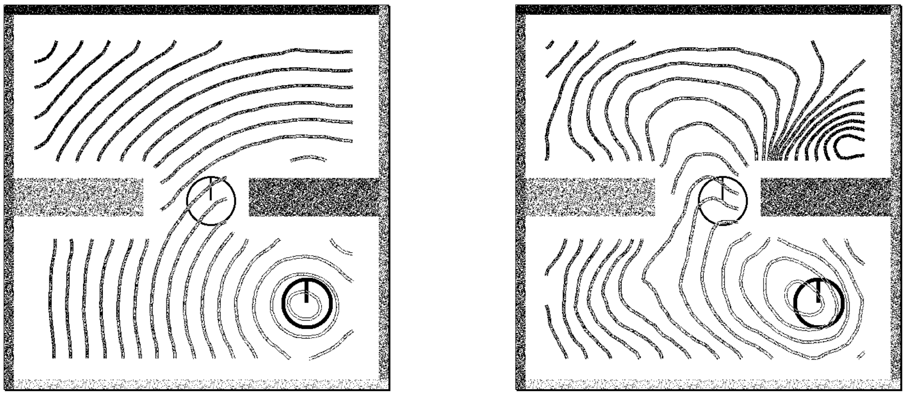

Figure 5.16. Contour Plots of the Outputs of the Homing Critic and the Combined Homing and Obstacle-Avoidance Critic for the Door Environment. The left panel shows a contour plot of the output of the homing critic. The right panel shows a contour plot of the sum of the outputs of the homing and obstacle-avoidance critic.

After training, the robot is able to form efficient homing trajectories starting from anywhere in the environment. Figures 5.17 and 5.18 show trajectories formed using only the homing critic. These demonstrate that the homing critic alone cannot adequately navigate home in the presence of obstacles. Figures 5.19 and 5.20 show trajectories formed using both the homing and obstacle-avoidance critics. The combination of the homing and obstacle-avoidance critics permits the robot to find a path home that is both efficient and avoids obstacles.

5.6 Discussion

The homing task represents a difficult control task requiring the solution of a number of problems. The first problem is that there is a small chance of getting home using random exploration. Unlike in the FSA environments described in Chapter , trial-and-error learning is intractable. The solution to this problem involves building a nominal homing critic that chooses straight-line trajectories home.

Next, because the state space is high-dimensional and continuous, it is impractical to uniformly place Gaussian units throughout the state space, and the training of networks with logistic hidden units is not easily achieved on this space. Instead we use

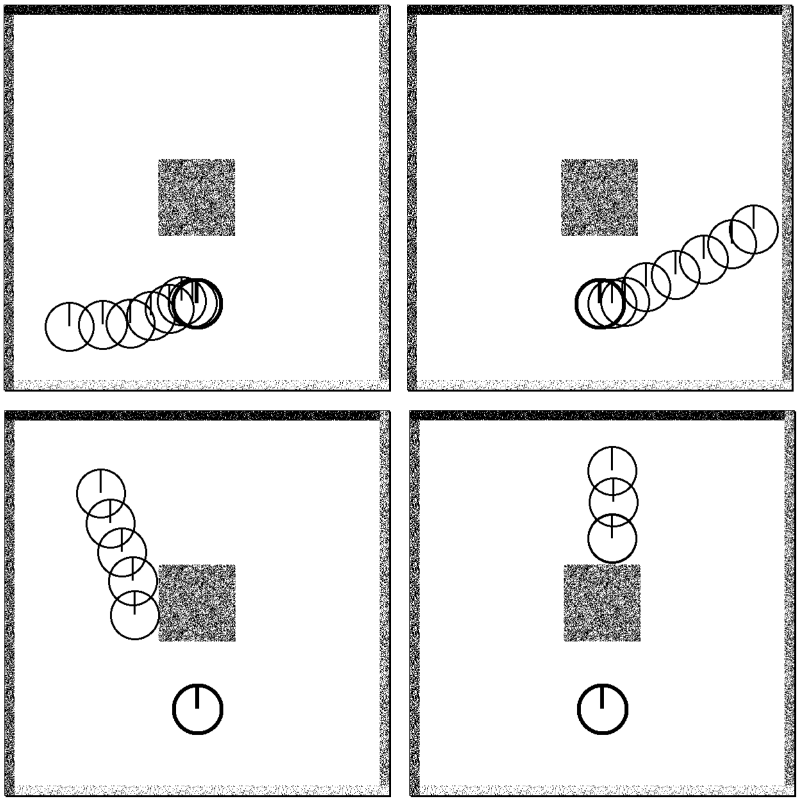

Figure 5.17. Trajectories Formed on the Obstacle Environment Using Only the Homing Critic.

Gaussian units whose initial weights are determined using expectation maximization, that in our case, creates spatially-tuned units.

Next, a model for the robot's environments is very difficult to learn. For this reason we used a hard-wired range-flow model whose performance is good in a wide range of environments. Here the philosophy is to learn only things that are difficult to hard-wire.

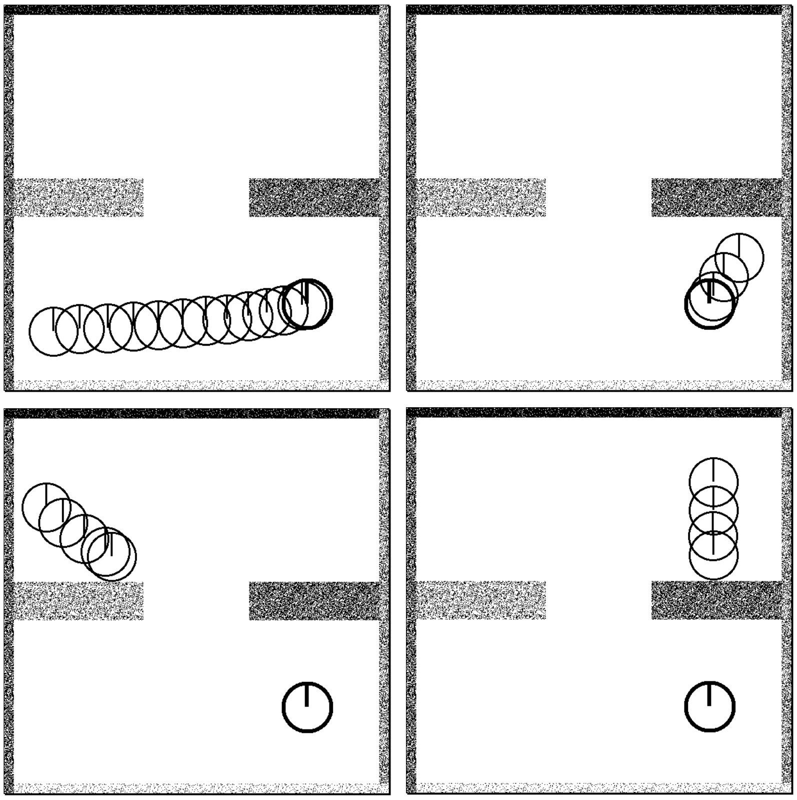

Figure 5.18. Trajectories Formed on the Door Environment Using Only the Homing Critic.

Despite our successful results, the current approach has many limitations. The training procedure is unrealistic for environments that are nonstationary because there is no mechanism for incrementally adjusting to a changing environment. A more practical training regimen would involve invoking the separate learning phases simultaneously. In particular, the Gaussian hidden units would be trained with an unsupervised procedure at the same time the homing and obstacle-avoidance critic were being trained.

Another limitation is that the obstacle-avoidance behavior does not generalize well to new environments. One way to fix this problem is to incorporate a hard-wired obstacle avoider into the navigation architecture. The hard-wired obstacle avoider would insure that the robot never bumped into obstacles and would then give the robot time to learn efficient goal-seeking avoidance trajectories (see Brooks [13]).

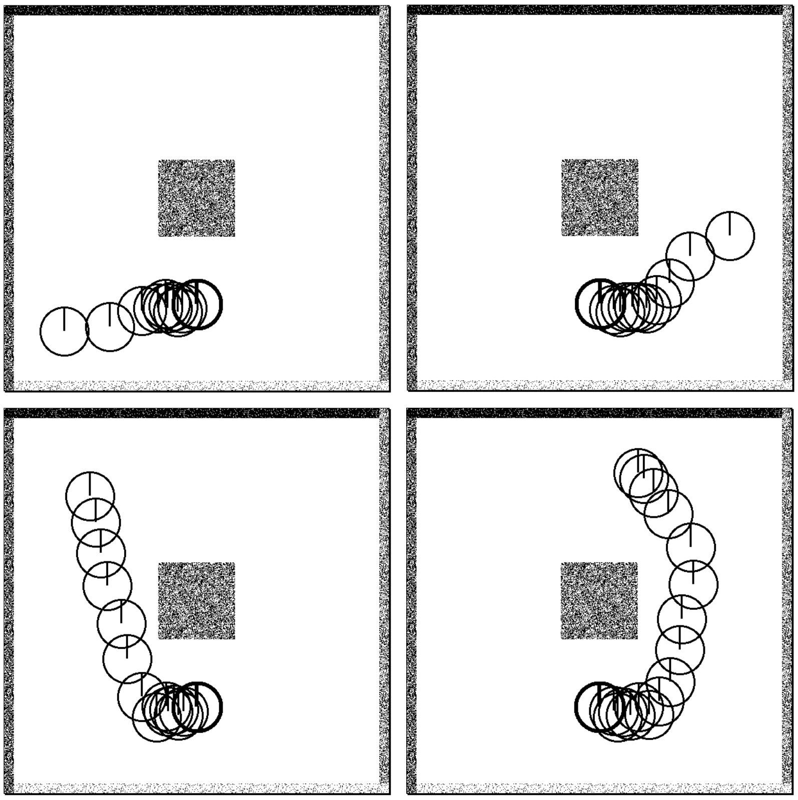

Figure 5.19. Trajectories Formed on the Obstacle Environment Using the Homing and Obstacle-Avoidance Critic.

The current training procedure will not always produce efficient trajectories. One reason is because it lacks a sophisticated exploration strategy. Without an exploration strategy, the robot will randomly choose a particular direction which might result in an inefficient trajectory. It would be easy to incorporate a simple exploration strategy like the one described in Chapter 3.7.

5.7 Future Work

There are many directions in which this work might be extended. First, we would like to apply this architecture to real robots using realistic sensors and dynamics. Real sensors are noisy and have a host of idiosyncrasies. An architecture must be robust in the face of sensor noise and must be able to accommodate the differences between individual sensors and the peculiarities of particular types of sensors, such as sonar. Connectionist networks are well suited for these problems for the following reasons.

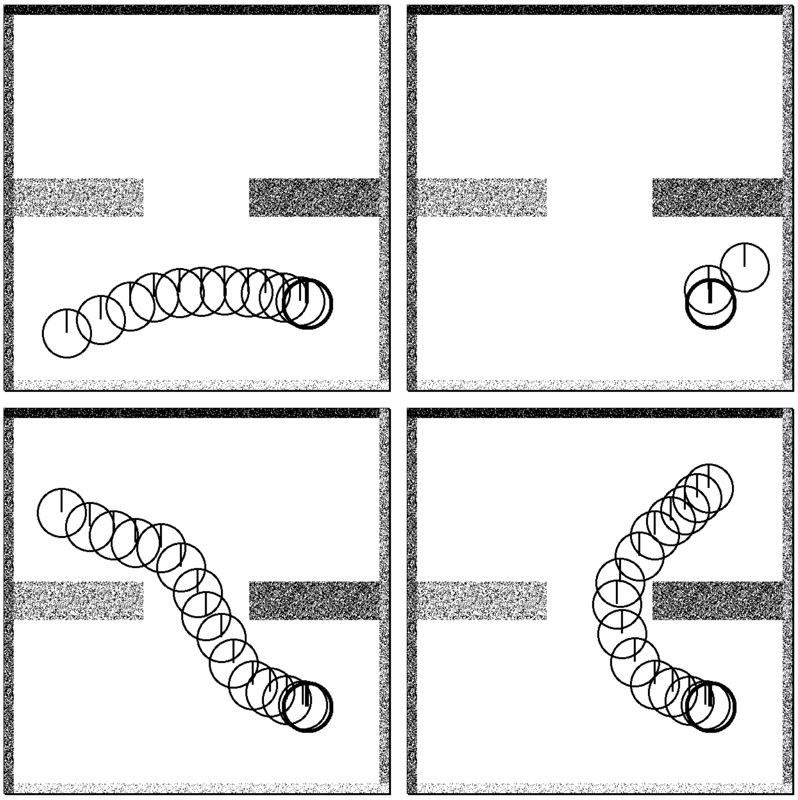

Figure 5.20. Trajectories Formed on the Door Environment Using the Homing and Obstacle-Avoidance Critic.

First, because of the statistical nature of connectionist learning rules, connectionist networks are able to learn in spite of noise in the examples, and they can also learn to approximate probabilistic functions in a principled fashion [31]. Secondly, the flexibility of connectionist networks as function approximators makes them suitable for modeling sensor idiosyncrasies.

We now present two alternatives for sensor modeling. The conservative approach is to add a few tunable parameters to our hard-wired range-flow model. These parameters could be adjusted during a calibration phase. The more radical approach is to replace the hard-wired model with a trainable network with a large number of hidden units. The range-flow model network would then learn the sensor idiosyncrasies.

In this chapter, we have not addressed the problem of dynamics. The basic training procedure is general enough to handle dynamics, but the range-flow model would have to be modified in order to account for the effects of dynamics. Barto, Bradtke, and Singh [7] have demonstrated a related approach to a robot control task involving dynamics.

To permit the robot to keep up with the real-time demands of real robots, it is important to improve the execution speed of the training and performance of the networks. One major improvement could come from caching the control law in a controller network instead of relying solely on the expensive gradient descent action selection search. The controller network could learn the action chosen by the gradient descent action selection search, and then the controller could be used without the search, once it reliably predicts the best action.

We also want to study long-range homing. Our current technique does not scale well with longer ranges for the following reasons. First, dead-reckoning is infeasible in larger environments, and thus a scheme must be developed for training the homing critic. We outline one such scheme in the next paragraph. Secondly, the TD method used to train the obstacle-avoidance critic does not scale efficiently. In order to effectively operate in larger environments, a robot must decompose its trajectories into subgoals. Recently, Jacobs [33] reported on an architecture for task decomposition, and Singh [90] has generalized this architecture for the learning of sequences of trajectories using TD methods. See also Schmidhuber [86] for related ideas.

While it is clear that a hierarchical mechanism is needed, it is not clear what form it should take. In navigation, there is a significant amount of a priori knowledge, such as knowledge of spatial relationships, that can be used to devise such a mechanism. One idea is to use the notion of hubs. In order to navigate from one place to another, a robot would navigate to the closest hub on the way, navigate to the closest hub near the destination, and then navigate to the destination. Another possibility is the idea[3] of using the metaphor of major highways and local streets. This scheme would involve navigating to a major highway using local streets, travelling on this major highway, and then getting off this major highway and navigating to the destination using local streets.

We would also like to investigate navigation tasks involving multiple destinations, as opposed to a single fixed destination. For example, the robot might be required to efficiently navigate to a variety of destinations that are either internally or externally specified. The obvious but inelegant solution is to use a separate critic for each destination. A better idea (due to Rumelhart [77]) is to have the homing critic learn to predict the x-y position of each view with respect to a particular fixed coordinate axis, instead of merely predicting the distance to the destination. In order to navigate to a particular destination, the straight-line potential function could be generated by the distance between the destination and the current position. The homing critic calculates these positions using their respective views. In short-range homing, dead-reckoning can be used to train an x-y homing critic. Instead of merely keeping track of the distance to the home position, dead-reckoning can keep track of separate x and y displacements from a given starting position.

In large-scale environments where dead-reckoning is not practical, another more sophisticated technique (due to Rumelhart [77]) can be employed. This technique utilizes an autoencoder as shown in Figure 5.21. The autoencoder learns to compute the identity mapping from views to views, with the constraint that the network's middle layer contains only two units. The learning procedure must construct an approximation to a two-dimensional invertible code for each view. The idea is to reconstruct the two degrees of freedom used to generate the views. The bottom half of the autoencoder could then be used as the homing critic. Figure 5.22 shows the results of training the autoencoder on a very simple environment without obstacles. It

[3] This is a classic idea from hierarchical planning (e.g., Sacerdoti [82] and Laird, Newell, and Rosenbloom [46]). Rumelhart [77] suggested this idea for connectionist navigation.

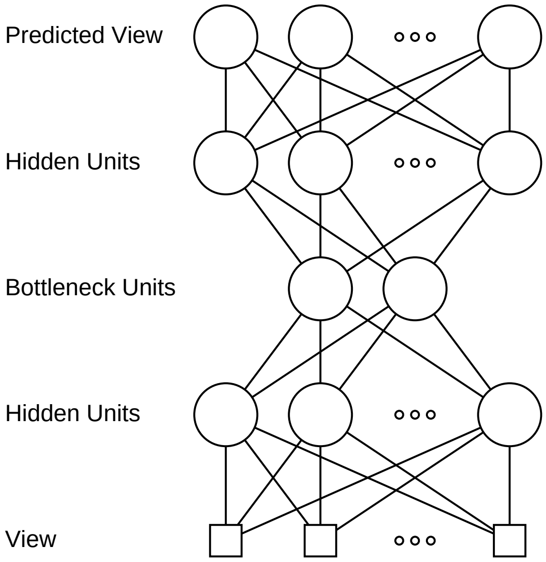

Figure 5.21. An Autoencoder Network for Learning a Two-Dimensional Code for Views. The network's task is to take a particular view as input and produce the same view on its output units using only two bottleneck units. One particular representation of views on the bottleneck units would be the x-y position where the view was generated. The autoencoder would then perform the transformation from view to x-y and then from x-y back to view. The bottom half of the encoder could be used to calculate the x-y position of a given view and could be used as a homing critic for multiple destination tasks as discussed in the text.

remains to be seen whether this technique will scale to more complicated environments with many obstacles.[4]

Addressing the problem of multiple destinations also introduces the problem of needing different obstacle-avoidance strategies for different destinations. Figure 5.23 shows an environment (called the culdesac environment) where there are two distinct destinations. Obviously, the obstacle-avoidance critic must have access to the particular destination in calculating its estimate.

---

[4]Rumelhart has suggested a couple techniques for finding better position codes on the bottleneck units. One technique is to add additional weight constraints and another is to start with more bottleneck units and to then gradually remove them.

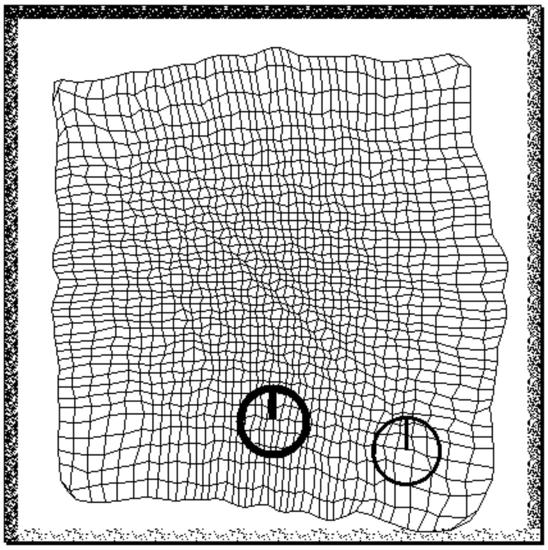

Figure 5.22. The Result of Training an Autoencoder on a Simple Environment Without Obstacles. This figure shows the responses of the two bottleneck units from Figure 5.21 to views generated at each of the feasible positions of a uniform 50×50 grid superimposed on the robot's environment. The responses of the bottleneck units to a particular view are plotted as an x-y point. Points generated by the bottleneck units for neighboring positions in the grid of views are connected by a line segment.

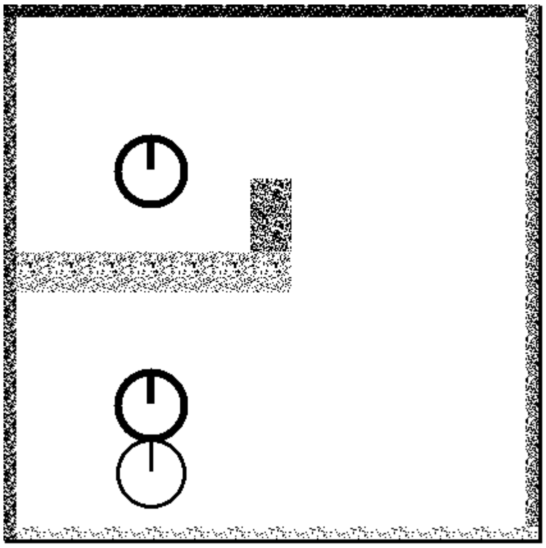

Figure 5.23. The Influence of Destination on Obstacle-Avoidance. Two different destinations are shown: one outside the culdesac and one inside the culdesac. In the first case, the robot should avoid the culdesac entirely, and in the second case, the robot should enter the culdesac.

5.8 Related Research

Our technique is most strongly related to navigation techniques that use potential functions. Our approach differs from these techniques in a number of ways. First, our approach involves learning. Our network is able to learn potential functions, automatically adjusting them to avoid both local minima and to produce efficient trajectories. In contrast, traditional potential function techniques suffer from the existence of minima and do not produce efficient trajectories. Secondly, our approach does not require object models because it learns potential functions using only sensory information. In contrast, the traditional technique involves building explicit object models representing the extent and position of objects in the robot's environment.

Our work is related to the subsumption architecture proposed by Brooks [13]. Both approaches involve the orchestration of multiple behaviors through the use of multiple experts. In our case, we examine obstacle-avoidance and homing behaviors. Also, neither approach utilizes explicit world models, but rather use only sensory information in choosing actions. Furthermore, neither technique utilizes classical planning, but instead use reactive planning, where actions are chosen based solely on the current view.

Our approach differs from the subsumption architecture in the following ways. The subsumption architecture involves an arbitration scheme which decides which behavior to invoke, while our technique has no such explicit mechanism. The various behaviors are invoked based on their contribution to the combined potential field. One could imagine more sophisticated techniques for combining behaviors within the potential function framework, such as the gating network proposed by Jacobs [33]. The subsumption architecture is based on finite state automata where the computations are performed primarily on binary signals, whereas our architecture involves continuous mappings constructed with connectionist networks. Finally, the subsumption architecture is handcrafted and involves no learning.

Recent work by Mahadevan and Connell [51] and Maes and Brooks [50] propose preliminary learning techniques for the subsumption architecture. Maes and Brooks present a reinforcement learning technique for learning an arbitration strategy that determines when to activate a hard-wired behavior. Mahadevan and Connell examine the learning of the behaviors given a fixed arbitration strategy. Singh [90] presents a technique that combines these two complementary approaches within the connectionist framework.

Mataric [52] reports on a subsumption architecture tailored for navigation. The method utilizes a set of simple, incrementally-designed behaviors that receive sonar and compass data. These behaviors cause the robot to wander around tracing the boundaries of the environment while detecting landmarks and creating a map of the environment. The map is a graph in which the vertices represent landmarks, and the edges represent adjacency relations between landmarks. Spreading activation is used to generate paths to the goal. The paths created using this technique are not necessarily efficient. They are composed of a series of boundary tracing trajectories that connect a series of landmarks leading to the goal position. In contrast, our technique utilizes a learning algorithm that produces time efficient trajectories. In practice, the trajectories formed by our navigation network are efficient but not optimal. On the other hand, our algorithm scales poorly, whereas Mataric's algorithm can be applied to large-scale environments.

We discuss two relevant and noteworthy connectionist navigation techniques. Pomerleau [69] describes an approach for training a 3-layer back-propagation network for road following. His system takes video images and range information as input and produces a steering command for the vehicle to follow the road. In contrast to our approach, Pomerleau's system is trained using a human teacher that provides steering commands as targets for the network. In our approach, we assume no such training information. The training regimen is completely unsupervised.

Park and Lee [64] present a technique for collision-free path planning using a connectionist network. They investigate the case of a polyhedral vehicle travelling through a set of polyhedral obstacles. For each obstacle, a penalty function is defined using a three-layer network. The total energy of a path is computed based on the collision penalty and the length of the path. Gradient descent is used to move the path such that the energy is minimized. In contrast to our sensory-based approach, their approach requires explicit object models. Furthermore, our approach is reactive, that is, it calculates its trajectory on the fly, whereas their technique requires a planning step at the outset.