BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M1

"Neural Networks for Mobile Robot Navigation"

by

John McCann

Neural Networks For Mobile Robot Navigation

John D. McCann

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science in the Department of
Computer Science at Brown University

February 1995

Professor Thomas L. Dean

Advisor

Contents

1. Introduction

One of the most exciting, and challenging areas of Artificial Intelligence is mobile robotics. Most mobile robotics applications depend on a control program, which receives inputs from various sensor equipment, processes that information to make a "best guess" as to the state of the world, or at least the immediate environment, and issues commands to the robot's navigation system based on the processing of this information. At this level of abstraction, the task of programming robots may sound trivial. As we move into a more concrete level, however, we begin to see the enormous complexities with which we are faced. There are several difficulties, not the least of which is making sense of our sensory inputs.

One of the most commonly used sensors in robotics is a camera. Computer vision is area of study all its own. From a mobile robotics programmer's point of view, we are interested mainly in how we can process the inputs from a camera in order to make our estimate of the state of the world. The problem becomes more complex if we add additional sensors to our system. The more sensors we add, the more information we have. However, processing this information can become very complex.

This report describes a system which uses machine learning to solve these complexity issues. We train artificial neural networks to learn specific functions. These functions in and of themselves are simple, but combined, they create a "brain" capable of complex behavior. The system is essential a fusion of different approaches to mobile robotics. There is some influence from the behavioral approach to mobile robotics, in that each individual neural network performs its function independently, while combined, they produce complex behavior.The goal is a modest one: to demonstrate that we can combine different approaches in vision and machine learning to build a practical robotics application.

2. Problem Description

The problem that we wish to solve is a fairly straightforward one. We want a mobile robot to be able to navigate through the hallways of an office environment, without stopping, and without colliding with any people who may happen to be in the hallways. We use a small robot equipped with a camera and sonar sensors. The control program will take the input from these sensors, process the data through a series of neural networks, and issue commands indicating the direction in which the robot should turn. The robot should navigate the hallways on a primitive level, that is, by following a particular wall. Upon detection of an obstacle, such as a person in the hallway, the robot should take appropriate evasion actions. This would normally involve turning towards the other wall, aligning itself with that wall, and continuing forward, or in some cases, turn around completely. Sometimes, the robot may encounter a dead end. In this case, the robot needs to be able to turn around and exit the dead end, without stopping. The robot should also be able to turn right-angle corners in our office environment.

We have three distinct behaviors that we wish our robot to emulate here. The simplest of these is the *wall following* behavior, in which the robot makes small angular adjustments at each iteration to keep itself moving parallel to one of the walls. The next behavior is the *avoid obstacles* behavior, in which the robot steers from one side of the corridor to the other, in order to avoid a collision with an obstruction in its path, presumably a person. The third behavior is the *turn corners* behavior, in which the robot makes right angle turns when it detects a wall in front of itself.

Each of these behaviors is controlled by a separate neural network. Individually, each network performs a fairly simple task. These networks are linked together by other networks which make decisions based on their outputs, and one piece of state information--the wall we are currently following. Together, these networks form a brain capable of guiding our robot through a hallway

environment.

3. Motivation and Related Work

As previously mentioned, this work is a fusion of different ideas. Pomerleau [5] developed the idea of using neural networks to train a robot to drive a car. It seemed logical that the technique could be adapted to fit the problem of a robot navigating the hallways of an office environment.

Camus [2] has developed a real time vision algorithm which calculates optical flow. Optical flow is essentially the calculation of the displacement of pixels over each frame iteration. This is used in our system to detect moving objects, such as people in the robot's path. Because it runs in real time, the optical flow algorithm is particularly useful for mobile robotics applications.

Horswill [4] uses a vision technique where a camera is aimed at the floor to detect *carpet blobs*, or areas where the carpet texture is consistent. Where the blob ends, there is likely to be an obstacle. This idea, fused together with Camus' algorithm forms the basis of the vision technique used in this system.

Another aspect of this problem that is worthy of study is the idea of sensor fusion. We combine sensor data from sonar sensors with the data obtained from the camera in our vision algorithm. Sometimes data from different sensors can be contradictory, in which case a control program must decide which sensor to believe.

There is also a flavor of the behavior approach to robotics in this work. The work of Brooks [1] and others has suggested that instead of building explicit, complex models of the world, a programmer should build up primitive instinct-like behaviors into a system that will eventually exhibit complex behavior. Our system does this to some extent, in that it relies on simple behaviors, and there are no extensive world models.

This work is essential a system derived from the techniques of Pomerleau [5], Camus [2], Horswill [4], and Brooks[1]. The result is a robotics application that uses artificial neural net-

works to learn from optical flow data, and sonar data, to produce a behavior based robot navigation system.

4. The Hardware

In order to carry out our experiments, we need the following pieces of equipment: a mobile robot with sonar sensors, a video camera, a framebuffer to process the camera data, a Sun Sparc1 workstation, a Sun Sparc10 workstation, and our hallway environment.

The mobile robot we have used is named Louie. Louie consists of a circular RWI Base, with a 30 cm diameter, and a square metal frame, about one meter tall. There are eight sonar sensors on the robot: two in front, two in back, and two on each side. We do not use the back sensors, so we deal with two sonars on each side, and two in front. Louie's base is capable of translating and rotating while in motion. Louie has no on-board computer. The robot is controlled by a forth board, which receives commands from a Sparc10 workstation. We tether the robot to the workstation through a serial cable.

The robot also has a small tripod mounted on top, onto which the video camera is mounted. The camera is connected by a video cable to a framebuffer, which processes the images from the camera. The framebuffer is controlled from a Sparc1 workstation. The Sparc1 machine sends the processed camera data to a process on the Sparc10 machine through a socket connection. The camera is plugged into a standard wall outlet, rather than the robot's battery supply. The camera is positioned so that it is aimed at the floor at an approximate forty-five degree angle. The data it collects consists of a 64x64 pixel image, representing the light intensities. This gives the camera a sixty degree arc of sight of the floor, approximately one meter from the robot.

The hallway environment used in these experiments is the fourth floor of the CIT building at Brown University. The carpeting and lighting in this hallway required some modifications in order to reduce noise in the camera data. A lightbulb was mounted on the robot, above the base, and below the camera. This acted as a headlight, and helped improve the robot's vision. The car-

pet also has a vaguely discernible pattern, which can create noise for the camera. Mats were placed in front of the robot, which contained a much more discernible pattern, to reduce noise in the optical flow algorithm.

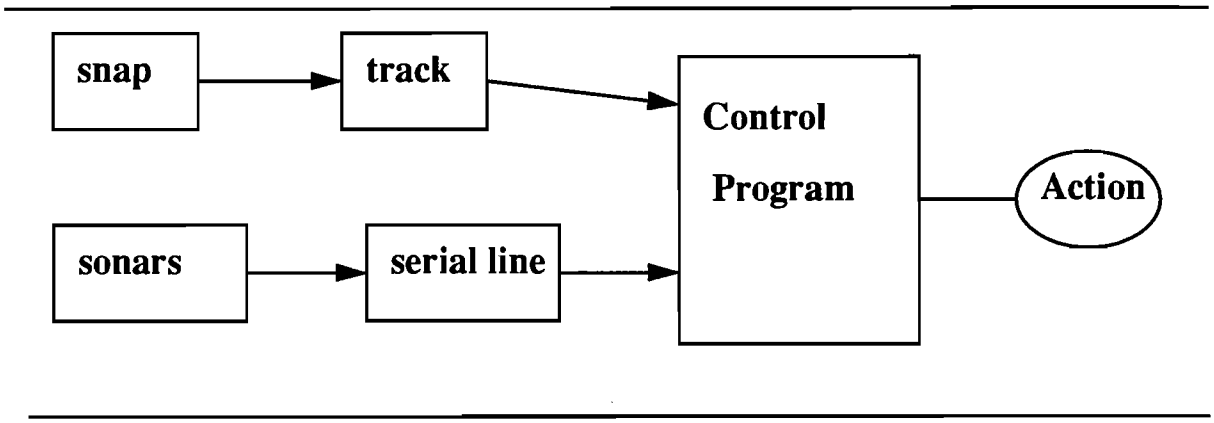

Figure 1: The processes which run our system.

5. The Software

There are three separate processes running simultaneously to gather the data and issue commands to the robot. One process is our main control program, which runs on the Sparc10 machine. The process which takes pictures with the camera runs on the Sparc1. The third process, which also runs on the Sparc10, is the optical flow algorithm.

5.1. The Optical Flow Algorithm

The optical flow algorithm is a process called *track*, developed by Ted Camus [2] at Brown University. The algorithm reads the image data from the process which takes the pictures, called *snap*. For each pixel in the 64x64 image, *track* compares the value to the values of the adjacent pixels. A bound is passed as an argument to *track*, 4 in our case. The bound indicates how many fractions of a pixel we want to consider in our matching. *Track* then finds the best match for each pixel, and creates an optical flow vector field, which is a 64x64 field whose values encode the pixel displacement per frame.

The encoded value is a short, unsigned integer, which is really two eight-bit fields. One field represents row displacement of pixels, the other column displacement. Given our bound of 4, the values for each field will range from -4 to 4, where a negative value represents a displacement up or to the left (North or West), and positive values indicate a displacement down or to the right (South or East). A value of 0 means that the pixel did not appear to change its position at all in that particular direction. A value of 1 means the pixel moved 1/4 of a pixel per frame. Values of 2 and 3 mean the pixel moved 1/3 and 1/2 pixels per frame, respectively. A value of 4 means the pixel moved 1 pixel per frame. It is this vector field which our control program reads in, and processes using an artificial neural network. For more information on the theory and implementation of the optical flow algorithm, consult [2].

5.2. Neural Networks As Decision Makers

The brain of our main program is a series of artificial neural networks. Each network was trained using the standard backpropagation learning technique [3] & [6]. Training sets were created by using combinations of real and program-generated data. Some of these networks require a fair amount of preprocessing. There are three primary networks, which receive data from the sensors as inputs. The other networks process the outputs of these primary networks and initialize the next iteration. The first of these primary networks is the optical flow network.

5.2.1. The Optical Flow Networks

As stated above, the *track* algorithm sends a 64x64 vector of encoded pixel displacements to the control program. These values form the basis of the inputs to our neural network. In order to speed up learning, and to reduce complexity, we scale down the vector field we receive by select-

ing only every other row and every other column. This leaves us with a more manageable 32x32 vector field. Since we are only interested in detecting fairly large objects, rather than identifying specific objects with precision, this reduction in complexity can be achieved without a loss of any critical information.

The general idea behind the optical flow algorithm is that objects that are closer to the robot will have more flow than objects further away. By aiming the camera at the floor, however, we would expect to see a even, consistent flow, since the floor is the same distance from the robot. A change in that even flow would represent the presence of an object other than the carpet the robot is traversing.

Some further modifications are necessary to make the data more useful. Since the floor is relatively close to the robot at all times, the optical flow algorithm tends to "max out" in the vertical direction. That is, the pixels appear to be shifting faster than one pixel per frame in the downward direction as the robot moves forward. This is analogous to driving a car: the immediate stretch of road the car is traversing seems to be moving (relative to the car) much faster than the trees and landmarks on the sides of the road. Given this fact, the vertical pixel displacement value is largely useless. We therefore only use the horizontal pixel displacement values as input to our neural network. The assumption is that if the robot is moving straight down a hallway, the pixels should not appear to be shifting in a horizontal direction at all. Hence, we would expect all horizontal values to be zero. Non-zero values indicate the motion of something other than the relative motion of the carpet, and thus a potential obstacle.

Since we are only interested in zero and non-zero values, we set each of the 32x32 inputs to 0 or 0.1, for zero or non-zero horizontal values respectively. The network has an output consisting of three bits, which are set to indicate whether or not the left, center, and right regions in front of

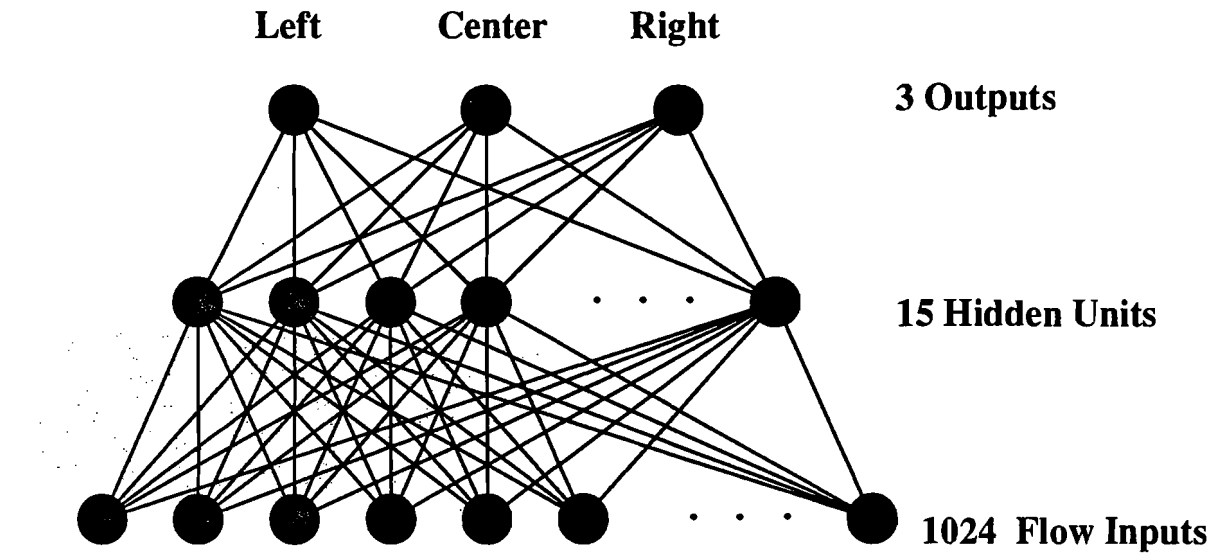

Figure 2: The Optical Flow Network

the robot appear blocked.

The training data for this network was gathered by repeatedly driving the robot over the carpet tiles, while walking in front of the robot at different positions, and storing the flow information. The outputs were manually added to the data files afterwards. Some data was generated by a program, which created a vector field with mostly zero values, but with concentrated areas of non-zero values, and the correct corresponding outputs. The real data tended to work better in the training sessions, because it was more generalized, rather than having rigid boundaries between zero and non-zero regions. A total of 150 examples were used in the final training set. The learning required 89 epochs for this set with 1024 inputs, 15 hidden units in a single layer, and 3 outputs.

The camera data can be susceptible to noise. Therefore, rather than pass the three outputs to the next level as they are, we instead take three readings, and pass their outputs to a network which basically employs a "best of three" strategy. If two of the three readings for a given region indicate a blocked path, then the result is 1, otherwise it is 0. These outputs provide a more accurate view of the path ahead for the robot.

Once we have the outputs of the flow network, we need to map that information to a direction in which the robot should turn. This is the job of the **flowdir network**. This network is much simpler than the flow network, and takes advantage of the fact that the flow net has reduced the camera data to a very simple bit field. The flowdir network has 5 inputs. Three of them are the outputs from the flow network. The other two parameters are values indicating which wall we are currently following. The variables **leftWall** and **rightWall** are initially set to 0 and 1 respectively. This means that we are initially following the right wall. These values are updated after each iteration. The larger of the values indicates the current wall.

The outputs for this network indicate a direction in which the robot should turn. The values are in the set **{60,40,20,0,-20,-40,-60}**, where a negative is a turn to the right, and a positive is a turn to the left, from the robot's point of view, for a total number of seven outputs. The angle with the highest corresponding output activation will be the angle that is recommended to be the robot's next turning command.

Given the limited number of parameters with which we must deal, it is easy to manually create the training set for this network. For example, leftWall and rightWall are 0,1 or 1,0, respectively. We also use 0's and 1's for the left, center, and right values. Using these guidelines, we create a set of 16 examples. We have 5 inputs, 7 hidden units, and 7 output units. These examples were trained over 235 epochs. The training set is designed to tell the robot to turn towards the wall it is

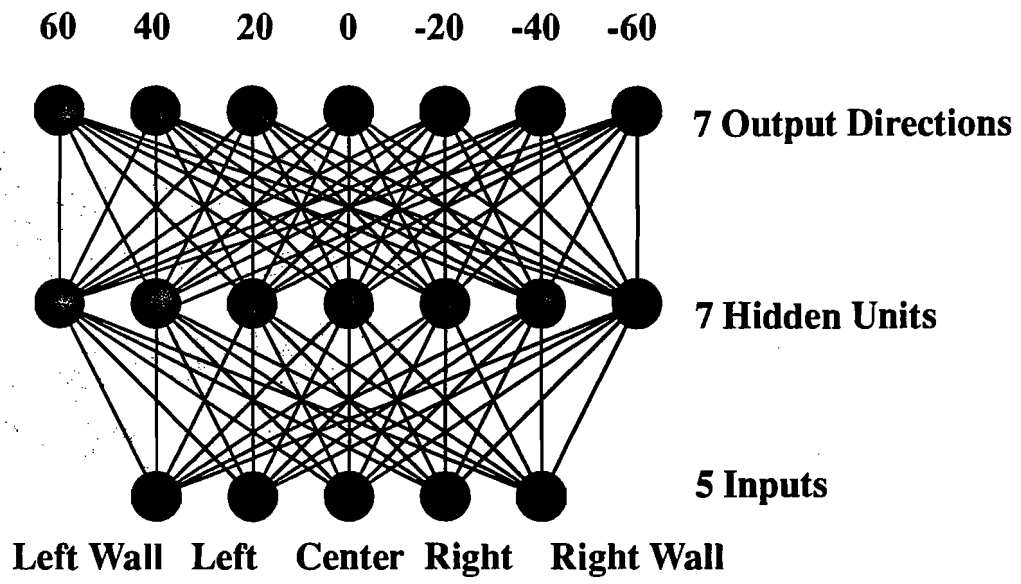

Figure 3: The Flowdir Network

not following if an object is blocking its path. The number of blocked regions in front of the robot, and the wall values, determine how much of a turn we wish to make. Together, the flow network and the flowdir network create an effective way to translate optical flow camera data into an action to be undertaken by a robot.

5.2.2. The Forward Sonar Networks

The camera is particularly useful for detecting moving objects in the robot's path. However, the optical flow algorithm is not as accurate in detecting smooth textured walls, which tend to exist in hallway environments. Also, the camera only sees a sixty degree arc in front of the robot. We receive no data concerning how close we are to the walls, or possible objects out of the

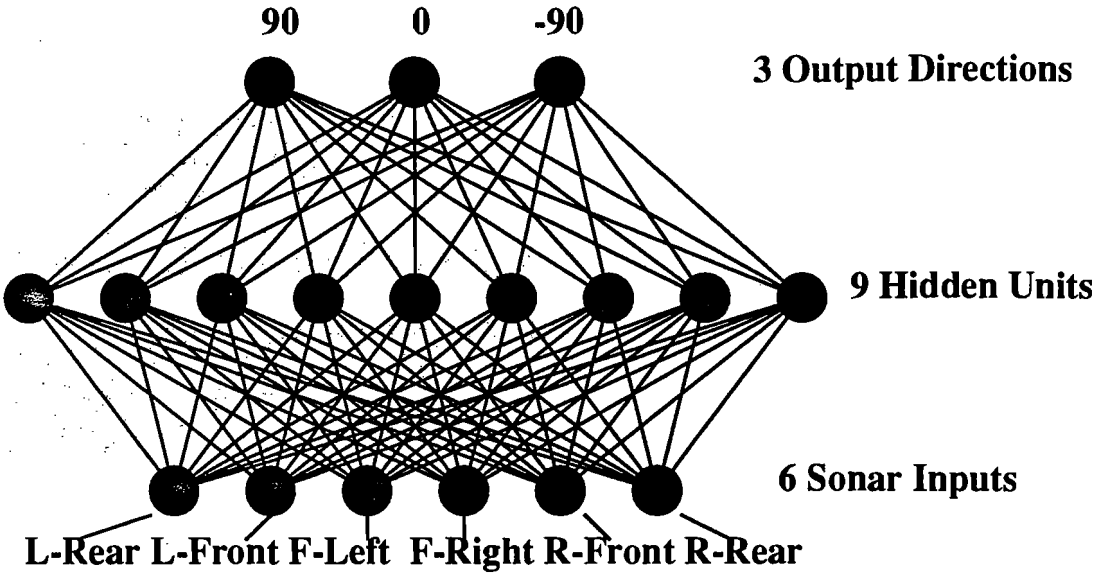

Figure 4: The Forward Network

camera's range of sight. This is why it is necessary to complement our camera data with data from the robot's sonars.

The sonars return a value, in mm, indicating the distance to the nearest object. The sonars tend to be most reliable when positioned perpendicular to the surface that we wish to detect. This is the premise upon which we created the **forward sonar network.**

This network takes as inputs the six sonar values starting from left rear, and going to right rear. The three outputs correspond to the angle set **{90,0,-90}**. The training set was generated by a program. Experimentation with the robot revealed the typical types of values one could expect to see, given different positions of the robot in a corridor. The program merely created sets of values that reflect these patterns, and assigned the appropriate output. A total of 7143 examples were gener-

ated, and trained over 44 epochs, with 9 hidden units.

The training set basically teaches the robot to go straight whenever possible, and turn ninety degrees only when there is no other direction in which to turn. One special case can occur which requires us to pass the outputs of the forward network to another network called the **fwddir network.**

Like the flowdir network, the fwddir network takes the output of the previous network, and adds the leftWall and rightWall values as additional parameters. The output is of the same size and type as the forward network. The special case which can occur is when the robot's forward path is blocked, but the left and right sides are equally clear. Rather than just randomly pick a direction, the fwdddir network lets the wall parameters determine the direction. In fact, in this case, the forward output array will activate both the 90 and -90 units. It is up to the fwddir to arbitrate between them. If there is no choice, then the output from the forward net will be output again. If there is a choice, the rule is to turn in the same direction as the wall the robot is currently following. This helps keep the robot moving in a path where it will continue to make progress, rather than turn back the way it came, as we will see in a later section.

The data set for this network was created manually, much like the flowdir network. A total of 8 examples were trained with 5 inputs, 4 hidden units, and 3 outputs over 28 epochs. The output from this network become the sonar recommendation as to which direction to turn.

5.2.3. The Sides Sonar Network

If there are no obstacles blocking the path of the robot, we would like the robot to move straight ahead. As the robot moves, however, it can gradually begin to drift off of its original heading, and even drift into a wall. For this reason, the robot needs to periodically turn slightly to

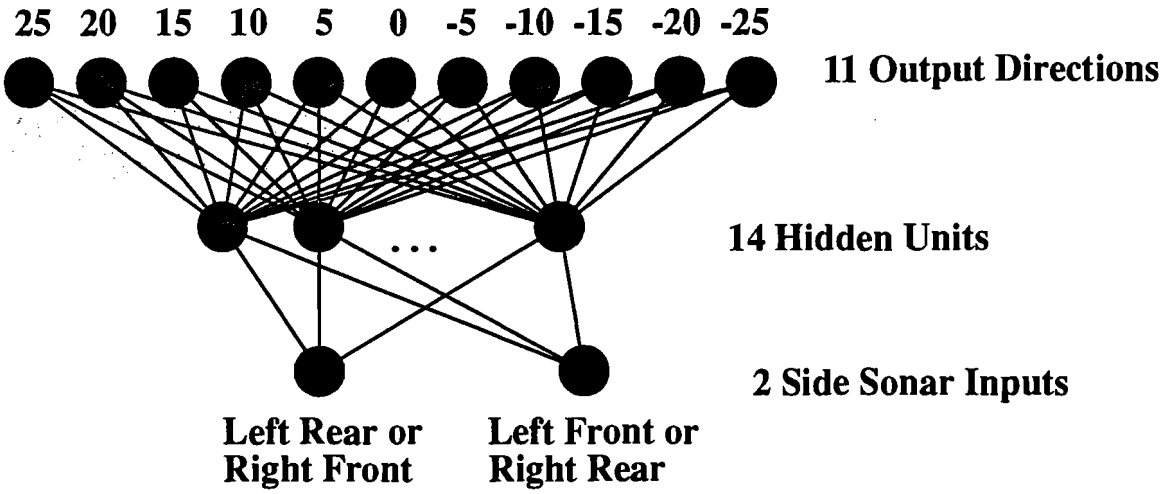

Figure 5: The Sides Sonar Network

keep itself on course. This is the motivation for *wallfollowing*, and the purpose of the **side sonar network**.

The side network takes two inputs, the two sonar values from the side of the robot corresponding to the wall we are currently following. The output array corresponds to the angle set **{25,20,15,10,5,0,-5,-10,-15,-20,-25}**. These angles represent the varying levels of adjustment the robot needs. Ideally, the range **{5,0,-5}**, should be the only active set, and indeed these are the most common outputs. The network is not concerned with which wall we are following, since the outputs will be the same regardless. The way to achieve this, is to pass the inputs to the network in two different orders. If we are following the right wall, the front side value is the first output, and the back side value is the second. If we are following the left wall, then the back side values is the first input, and the front side value is the second. This allows the network to be independent of the

wall being followed, and eliminates the need for another complementary set of training examples.

The training of this network proved extremely difficult. There are two reasons for this. First, there are a large number of outputs to add to the complexity of the network, much more than any of the others. Secondly, and more importantly, the selection of the correct output to activate depends upon very small differences to the input values. With the forward network, we were dealing with distances that varied in magnitudes of tens of centimeters. The side network deals with values with differences less than, sometimes much less than, 10 cm. The combination of the large output set and the vaguely distinguishable inputs made convergence in training a serious problem. The solution was to restrict the training set to a small number of examples, with one value fixed at 100mm, and the other changing up to 225mm. We also scaled the values with the formula: *n*4-300*, to make convergence easier. This allows the network to learn the only thing that is it critical that it learn: to assign outputs based on the difference between the two input values. We created a total of 36 examples, with 14 hidden units over 667 epochs.

The training set was designed with the notion that for every 25mm of difference between the two values, there should be a turn of 5 degrees. The direction is determined by the ordering of the inputs. With this network in place, we have used the last of our sensor information, and we must decide which of the three primary networks should be followed.

5.2.4. The Integration Network

The **integration network** is basically an arbiter, which decides which network's output is the correct one to follow. The 21 inputs values consist of the 3 outputs from the fwddir network, the 7 outputs from the flowdir network, and the 11 outputs from the side network. The output is a three bit field which corresponds to the fwddir, flowdir, and side networks respectively. The output with

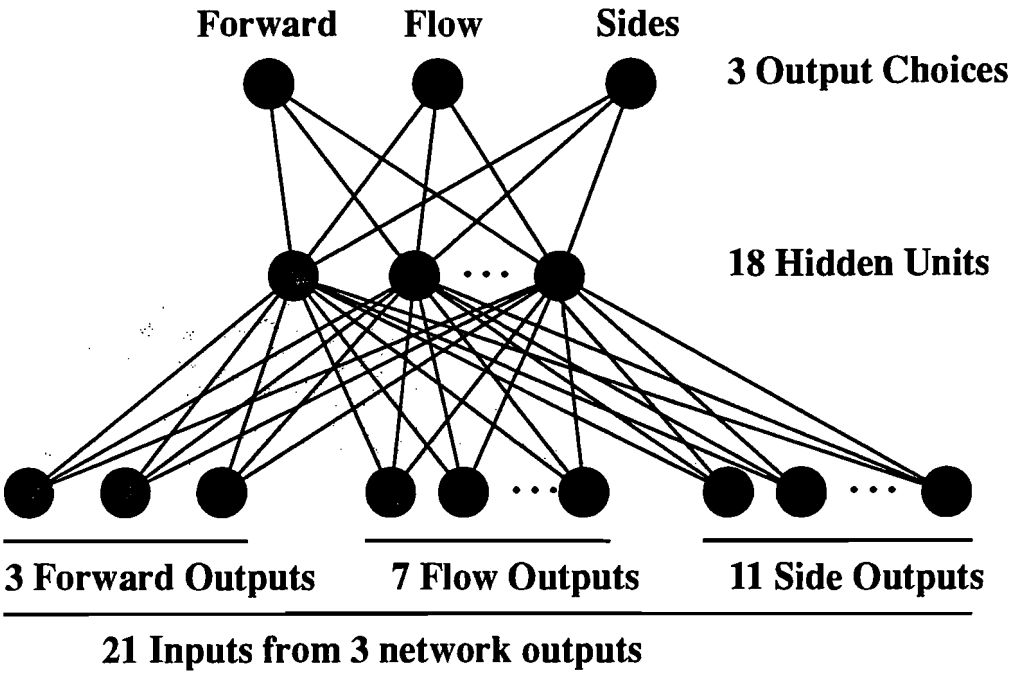

Figure 6: The Integration Network

the highest activation is the network that will be followed.

The training set was computed by a program, which generated patterns of outputs from the three primary networks. The output is designed to reflect a simply hierarchy: 1.fwddir, 2.flowdir, 3. sides. The fwddir network has highest priority, since it is recommending a right angle turn to presumably avoid hitting a wall. The flowdir has next priority, since it is recommending a course change to avoid an obstacle in the robot's path. The side network has the lowest priority, since avoiding collisions with walls and obstacles is more important than alignment with the wall.
If the fwddir net has any output other than its 0 degree (go straight) output active, then the integration activates its first output node. If the fwddir says to go straight, it is up to the flowddir. If the

flowdir has any active output other than its 0 degree output, then the integration network activates its second output. If both fwddir and flowdir say to go straight, then we just want to align the robot with the wall, so the integration network activates its third output. The 0 degree turn outputs for the fwddir and the flowdir basically enable the next priority, while the other outputs act as disablers. The program generated 14,784 examples, which were trained with 18 hidden units over 49 epochs.

5.2.5. The Wall Networks

Once we have decided which output value to follow, we must prepare for the next iteration of sensor readings. We must update the values of leftWall and rightWall, in the event that a course change has affected the choice of which wall to follow. We have two **wall networks**, one which corresponds to the fwddir network, and the other to the flowdir network. The inputs to both networks are the current values of leftWall and rightWall, and the outputs of the corresponding sensor direction networks. The two outputs are the new values for leftWall and rightWall, respectively. Both networks were trained with manually constructed data, with boolean values representing the possible inputs and outputs. The fwddir wall network was trained with 6 examples, and 4 hidden units, over 16 epochs. The flowdir wall network was trained with 14 examples, and 5 hidden units over 14 epochs.

The first wall network is used if the direction is chosen from the fwddir network. The rule that is learned for this network is: when making a 90 degree turn, the wall to follow to the opposite of the direction in which was just turned. This, combined with the rule for choosing which 90 turn to make when there is a choice, insures that the robot will continue moving in the same forward direction, rather than begin backtracking.

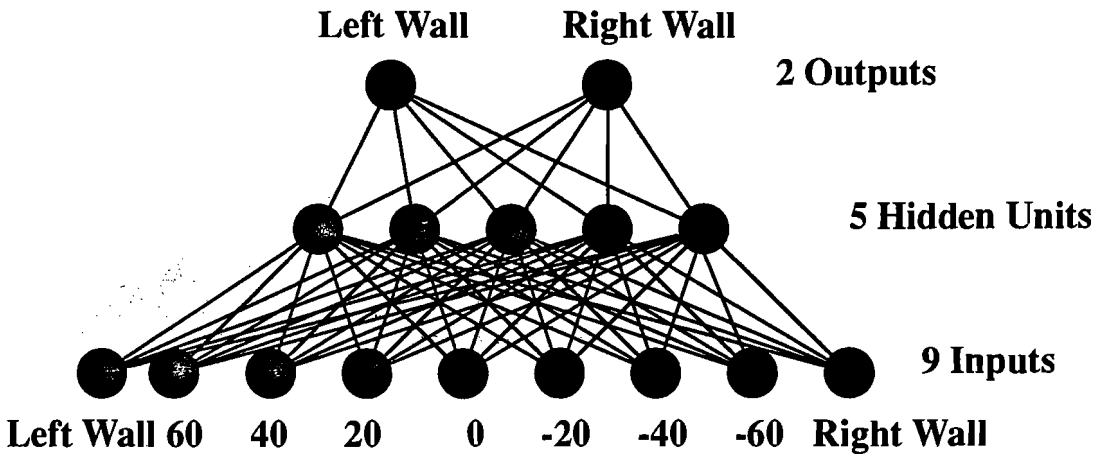

Figure 7: The Flow Wall Network

The second wall network, which corresponds to the flowdir output has a simpler rule: follow the wall in whose direction you've just turned. For example, if the robot veers 60 degrees to avoid a person walking down the hall, and had been previously following the right wall, then the left-wall would be the wall to follow for the next iteration.

5.2.6. The Delay Network

In the event that we choose the flowdir network's output as the robot's direction in which to turn, we need to allow enough time for the robot to get to the other side of the corridor, before taking the next set of sensor readings. We need a variable delay, which is determined by both the angle in which the robot has turned, and the two side sonar readings facing the wall towards which we are turning.

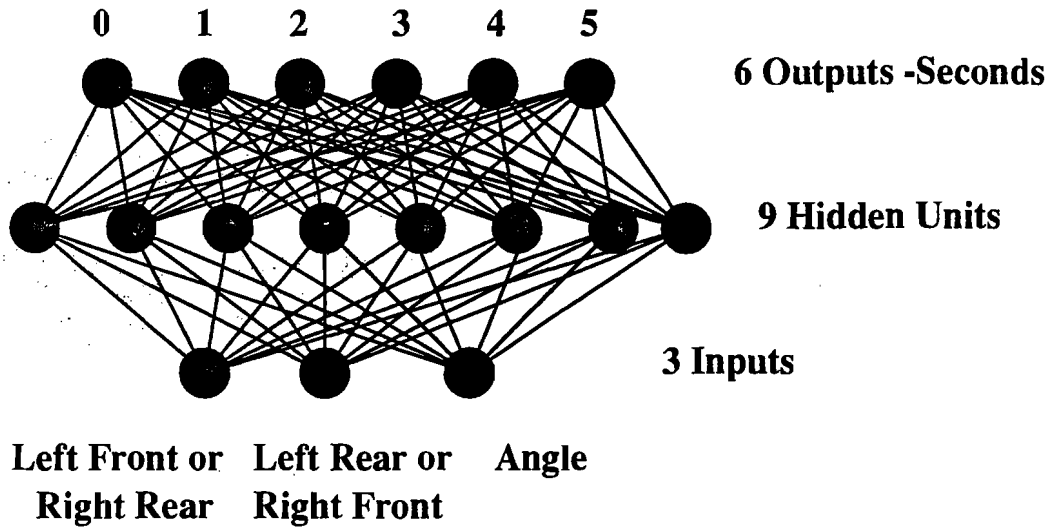

Figure 8: The Delay Network

The inputs for the **delay network** are the above mentioned angle and two sonar readings. The output array corresponds to the number of seconds the control program should wait before starting the next iteration. The first output corresponds to a wait of 0 seconds, the next 1 second, and so on, up to 5 seconds.

The training set for this network was manually created to reflect the obvious rule: the greater the distance, the more delay. The angle also plays a factor: after a sixty degree turn, the time required to reach the other side is less than after a 40 degree turn. Usually, 20 degree turns do not cause a shift to the other side of the corridor, so this limits the delay considerably. The training set of 27 examples were trained with 8 hidden units, over 778 epochs.

5.2.7. Evaluating the Usefulness of Neural Networks

Now that we have seen each of the neural network components which make up our system, one might ask, "Why do we need all these networks? Would it not be easier to just program their functions directly, in a conventional manner?" In this section, we will address this issue.

In the case of processing the flow data from the *track* program, a conventional program would need to check each pixel displacement value, keep track of where non-zero values appear, and in what regions they appear most often. It would also need to identify concentrated blocks of non-zero values, since these would indicate the presence of obstacles. The neural network accomplishes all of this in the training phase. All that is required is a diverse enough training set that allows the network to train on all possible types of examples. It was fairly simple to construct such a set. Adding the output values to the file after gathering the input data was time consuming, but very easy to do. Another issue is the flexibility of the network. A conventional program would at some point need to specify some rule for arbitrating between two actions. If, for example, the program were to require *n* values in a particular area to be non-zero, in order for that area to be considered blocked, what happens if there are *n-1* non-zero values? Conventional programming techniques tend to have an *all or nothing* approach when it comes to this type of situation. The flow network does not require as much rigidity in its specifications, and is therefore able to generalize more, on a case by case basis. This is similar to the problem of training with programmed data versus real data which was discussed in section 5.2.1. Given the requirements necessary to program the processing of the camera data, it seems that neural networks have definite advantages over conventional techniques.

The advantages may not be as obvious in the cases of processing the sonar information. There are much fewer inputs values to process, than with the camera. In the case of the forward behav-

ior, a conventional program would need to check the forward sonars values to determine whether they are large enough to determine that it is safe to continue to move forward. If not, it would have to check each side, and determine in which direction it is best to turn. These are essentially the same rules used in the program which generates the training examples for the forward network. The advantage of using the network approach is again in the generalization. Suppose that we decide that in order for our control program to decide it is safe to move ahead, both forward sonar readings must be above 500mm. Suppose one of the values were 501mm. The conventional program would say it is safe to go straight, while the forward network, would correctly generalize, and determine that it is not safe. The actual training of the network was trivial once the examples were generated. Given this, the network approach seems to offer an advantage that conventional programming does not, with minimal cost. The exact same argument can be made in the case of the sides network. For roughly the same amount of work, we gain much better generalization by using networks over conventional programming.

The other networks are fairly trivial in their architectures and functions. It would be fairly straightforward to program their behaviors into a control program directly. However, it was just as trivial to create and train the small training sets these behaviors require. The one exception would be the integration network, however. It may have been much easier to simply check the maximum output value of the three primary networks, in order to determine which one to follow. In this case, we do not need to generalize, so we would lose nothing by applying a direct, rigid rule.

In general, the decision of whether to use neural networks, or standard programming techniques, depends on the nature of the behavior being programmed. It seems that the larger the set of input data is, the more likely neural networks will be easier to use than standard techniques. The decision also depends on how generalized the behavior must be. The more generalization that

is required of the system, the more likely neural networks would be useful. Certainly, in the case processing camera data, the usefulness of neural networks becomes obvious.

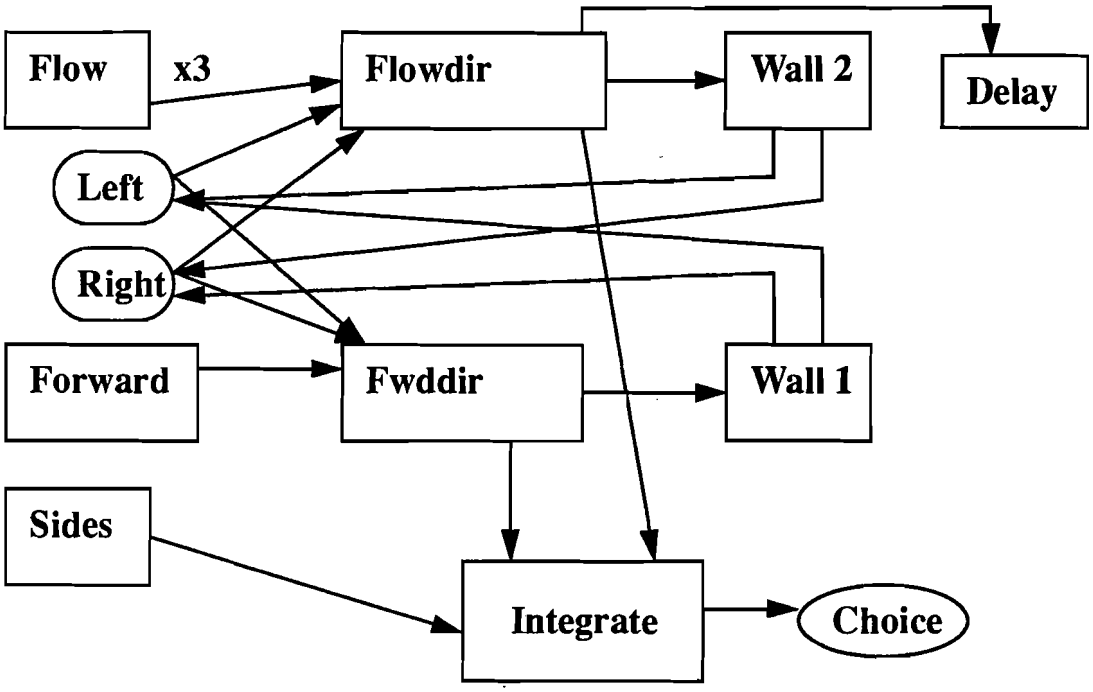

Figure 9: Interaction of the Neural Network System

5.3. The Control Program

Now that we have seen the various neural networks, and their specific functions, it may be helpful to see just how the main control program uses this collection of networks to make intelligent decisions as to the actions of our mobile robot. The first thing we need to do is initialize everything.

We first run the *snap* program on the Sparc1 workstation. This controls the frame buffer and the camera. The best flow results seem to come from setting a delay of 100 milliseconds between each picture. This means ten pictures, or frames, are taken each second by the camera and processed by the frame buffer. The *track* program must also be running next. This computes the optical flow for the camera data, and writes it to a socket connection. The first step, then, in the *driver*

program, is to establish a socket connection for the output from *track*. Once the connection is made, we need to make a connection to the robot. We then initialize leftWall to 0 and rightWall to 1. We then set the robot into motion at the constant speed of 5cm/sec. This speed, combined with the camera frame delay rate, seems to produce the best optical flow. We are now ready to begin the main control loop.

The first thing the loop does is read in the optical flow data from the socket connection. Three separate flow vector fields are read in. Each of these are propagated through the flow network, which reduces them to the three bit field, indicating whether or not each region is blocked. The three output bit fields are combined into one by propagation through the "best of three" network. We now have the camera information in a form we can use.

We then need to read the sonar values. The sonars are fired 3 times, and their readings averaged together. This reduces noise and misreading that occasionally occur with the sonars.

We then propagate our sensor readings through their respective neural networks. The six sonar values are propagated through the forward network, and then through the fwddir network, along with the leftWall and rightWall values. The results of the flow networks are propagated through the flowdir network, along with the wall values. The side sonar values are scaled to be in range of the training data. The smaller of the two side values os scaled to 100mm, and the other value is scaled so that the difference in the values is the same as the original. The modified values are then propagated through the side network. We now have all possible choices for the turning direction.

We then propagate the outputs of the fwddir, flowdir, and side networks through the integration network. The output from this network tells us which of the three choices to follow. We then issue the command to the robot base to turn the appropriate angle. If the fwddir or flowdir choice is chosen, then we call the corresponding wall network to set the leftWall and rightWall variables

for the next iteration of the loop. If the flowdir is chosen, we also call the delay network, to get the appropriate delay time. At the end of the delay, we turn in the opposite direction that we just turned, in order to realign the robot with the new wall. If flowdir is not chosen, then we use a fixed delay of 1 second.

While the control program is deciding what to do, the *snap* and *track* processes are continuously taken pictures, calculating flow, and piping it via socket connection to our control program. To insure that the program is receiving the latest flow information, we need to read away all of the flow that is accumulated during any delay periods. We are now ready to begin the next iteration of the loop. The loop continues until aborted by the user.

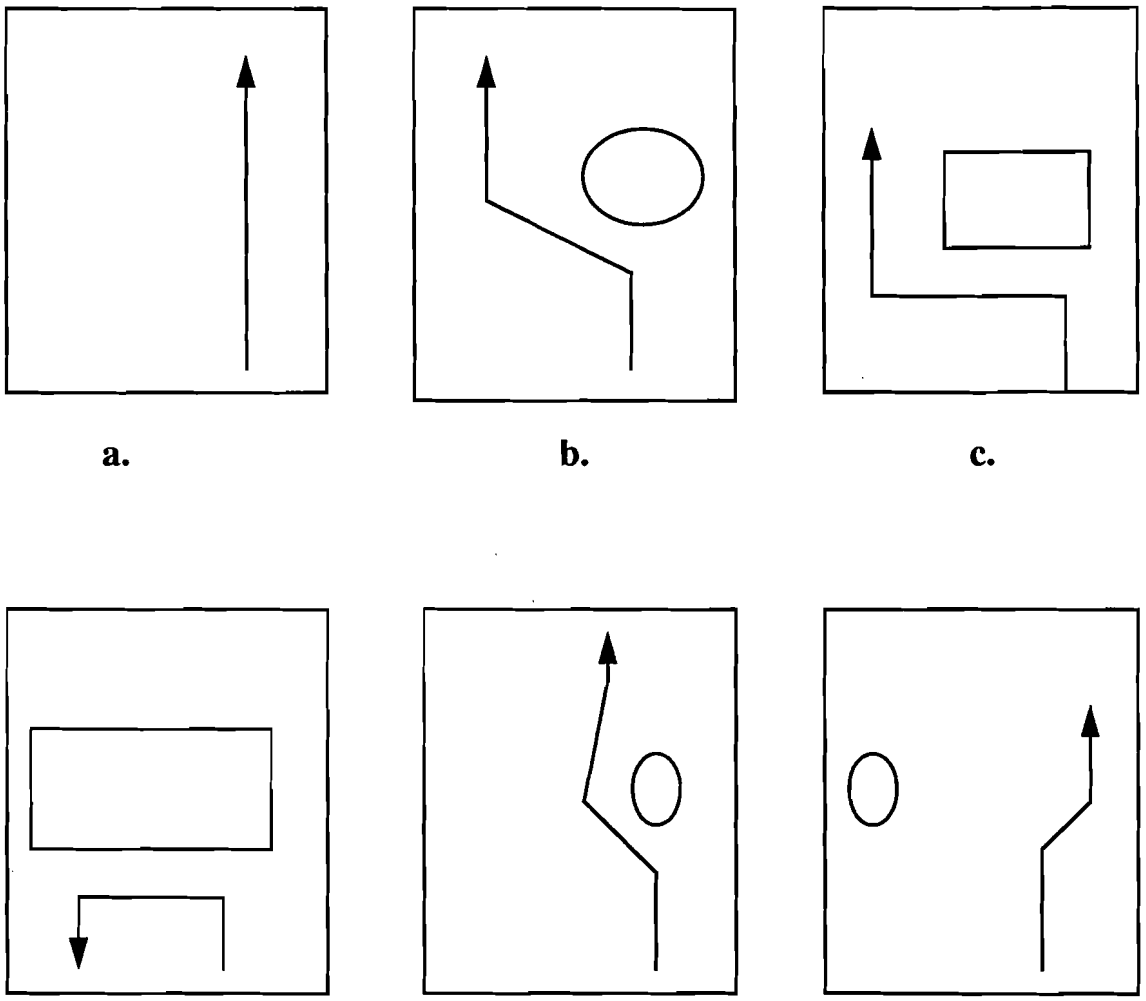

Figure 10: Paths taken by the robot.
a: Straight down the hall.
b: Move towards other wall to avoid moving object.
c: Make two right turns to avoid non-moving object.
d: Make two right turns to avoid dead end.
e: Veer away from wall to avoid small obstacle.
f: Veer towards wall to avoid obstacle on other side.

6. Experiments and Results

The first, and simplest test we tried on the robot was to have it move straight down the hall-

way, without any obstacles. Ideally, the flow algorithm should detect a smooth consistent vector field on the floor, and the sonars should return large distances for the front, and much smaller ones for the sides. Each sonar value on each side should be nearly equal. Only small adjustments from the side network should be changing the robot's direction.

This test basically worked out fine. The robot travelled to the end of the pattern tiles on the floor, making only slight course corrections. Any failures could be attributed to anomalous readings in the sensors. Occasionally, the camera would record flow where there should be none. The sonars would occasionally give strange readings. Our noise control strategy of taking three flow readings, and three sonar readings, reduced the overall noise considerably, however.

The next test was a bit more ambitious. We wanted Louie to be able to detect a person walking slowly towards it, veer out of the way, and swerve back to a forward course before hitting the opposing wall. The robot should turn 20, 40, or 60 degrees, depending on how many regions are deemed blocked by the flowddir network. The delay should bring the robot at least beyond the center of the corridor, before allowing it to turn back to face the forward direction again.

This test was basically successful. When walking directly in the robot's path, the entire flow field became active with horizontal motion. So the robot made its 60 degree turn. The delay computed by the delay network brought the robot to within 30 cm of the other wall, before turning back. On the next iteration, the side network was able to more accurately align the robot with the new wall. When walking to one side of the robot, close to the same wall, the robot made a 40 degree turn, as expected, and realigned itself on the other side accordingly. When walking sideways, hugging the wall, the robot was able to make a 20 degree turn. With a twenty degree turn, the robot does not shift sides in the corridor. Rather, it moves away form the wall just enough to get by an obstacle up against the wall, which was the person walking. The robot was able to pro-

duce these desired results. We were also able to repeat the test several times; the robot was able to shift back and forth between walls and align itself.

Another similar test was to see how the robot would react if it were following the right wall, and a person walked down, following the left wall. If the robot was close to the middle of the corridor, then it would turn 20 degrees towards the wall it was already following, to move in closer. The delay would again bring the robot to a close, but safe, distance from the wall. If the robot was already close to the wall, the robot and person were able to pass one another without a course change.

These tests were successful, as long as the camera data was relatively free of noise. One condition that can cause problems is if the person is wearing uniform colored pants that are of either a very light or a very dark color. A uniform color across the camera's field of vision can neutralize the optical flow. Non-uniform colored pants such as stonewashed jeans tend to produce the best results.

The next tests involved right angle turns. We wanted the robot to be able to detect a non-moving object, one that perhaps generates little or no optical flow, and turn at a right angle to avoid it. We would then expect the robot, on the next iteration, to detect the wall in front of it, and turn again. If the initial obstacle is now out of the way, the robot should choose to turn back to the direction in which it was previously traveling. If one side is blocked, presumably as a result of the presence of the object which caused the initial turn, the robot should turn in the other direction, presumably making a U-turn out of what would appear to be a dead end.

These test were quite successful, but relied on two modifications; one hardware, the other software. The problem was that the walls in our hallway environment are not perfectly smooth. They have small bumps and ridges. With the relatively poor lighting available in the hallway, these

ridges can create shadows which the optical flow algorithm interprets as flow. The camera sees the wall, before the sonar readings are close enough for the forward network to turn to robot at a right angle, and the robot turns 60 degrees instead of 90. The hardware solution was to mount a "head-light" on the robot. The lightbulb provided enough light to eliminate the unwanted shadows that were creating the noise in the optical flow algorithm. The light must not be too bright, or it will make normal object detection very difficult with the optical flow algorithm.

The other problem was the robot often chose to make a U-turn, even when it could have turned back on its original heading, and resumed a forward course. The software solution was to bound the side sonar values by a maximum limit of 1000mm. If the side values exceed these maximum bounds, then they do not provide any useful information for wall alignment; the sonars could not be detecting a wall in our hallway with those distances. By binding the values, the side readings were both sufficiently large when facing a wall, that it chose the correct 90 turn to make. Any obstacles would be within the bounded range, and would be detected by the sonars.

7. Future Work and Conclusion

Although the implementation of the general ideas of this project were successful, there is always room for more to be done. Sensor noise is a problem that has plagued mobile robotics programmers for many years, and will probably continue to do so. One enhancement could be to try and eliminate more noise.The camera often produced noisy readings, given the lighting conditions in the hallway. One might try to improve the lighting conditions, or explore ways to reduce the camera's sensitivity to shadows.

Another problem that could be addressed is the issue of speed. The robot moved fairly slowly, only 5cm/sec. One could explore ways of increasing the robot's speed while still receiving reasonable data from the optical flow algorithm.

One might also try using different equipment, and different environmental conditions. The system would be more robust if it worked on any type of carpet or floor surface. The robot, Louie, which was used in these experiments, often had problems. Sometimes, as a day of testing went on, the robot's batteries would run low, resulting in a speed slower that expected, which threw off delays and optimal turning points. The system was built with the intent of using this particular robot, in this particular type of environment. One may want to generalize the system, so that it will work for robots of different sizes, shapes, and possibly even different sensors. The sonars sometimes would return anomalous readings, which would throw off a testing session. More sonars clustered together, might overcome these problems, and provide more accurate data.

Despite these constraints, the system was able to accomplish what was expected of it. We have demonstrated a practical application of the optical flow algorithm. We have shown that different types of sensor information can be combined to allow intelligent control decisions to be made by a mobile robot. We have seen that neural networks can be trained off-line, and then used to pro-

duce real time outputs to the input sensor data. We have also demonstrated that neural networks are capable of performing simple low level functions, which combine to produce a complex behavior. This system is not completely behavioristic by any means. However, the neural network configuration is not unlike a behavioral approach. In any event, the system clearly demonstrates the usefulness of machine learning, and neural networks in particular, in the field of mobile robotics.

8. Bibliography

[1] Rodney A. Brooks. The Whole Iguana. Robotics Science, The MIT Press, Cambridge. 1989.

[2] Ted A. Camus. Real-Time Optical Flow. PhD Thesis, Department of Computer Science, Brown University, 1994.

[3] John Hertz, Anders Krogh, and Richard G, Palmer. Introduction to the Theory of Neural Computation. Addison-Wesley Publishing, Redwood City, CA, 1991.

[4] Ian D. Horswill. Specialization of Perceptual Processes. PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1993.

[5] Dean A. Pomerleau. Neural Network Perception for Mobile Robot Guidance. Doctoral Thesis, School of Computer Science, Carnegie Mellon University, 1992.

[6] Elaine Rich and Kevin Knight. Artificial Intelligence. McGraw-Hill, Inc. New York. 1991.