A Modular Neural Network Approach to Autonomous Navigation

Ian Lane Davis

CMU-RI-TR-96-35

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

April 1996

This research was sponsored by the U.S. Bureau of Mines under contract H0358021 "Robotic Sensing and Control for Underground Mining."

Carnegie Mellon

Robotics

Thesis

A Modular Neural Network Approach to Autonomous Navigation

Ian Lane Davis

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in the field of Robotics

ACCEPTED:

Anthony Stentz — Thesis Committee Co-Chair	5-17-96 — Date
Melvin Siegel — Thesis Committee Co-Chair	96 Jun 17 — Date
Matthew T. Mason — Program Chair	July 10, 1996 — Date
Raj Reddy — Dean	7/11/96 — Date

APPROVED:

Paul Christiano — Provost	12 July 1996 — Date

This thesis is dedicated to my wonderful wife, Sandy, for her infinite support and patience, and to my parents, Jean and David Davis for starting me on the road to a Ph.D. and to life in general.

The work contained in this document was supported in part by a National Science Foundation graduate fellowship.

The author would like to acknowledge the help and contributions from many people, including, but surely not limited to the following:

At the Field Robotics Center: Alonzo Kelly, for providing many of the software tools that proved useful in my Navlab II research as well as the benefit of his extensive experience with and knowledge of the Navlab II vehicle; Jim Frazier for keeping the Navlab II up and running (a gargantuan task); R Coulter, Gary Shaffer, and Barry Brumitt off of whom many useful ideas were bounced during their times as my officemates; Jim Martin for keeping me in an office; and the whole of the FRC research, systems, and computing staff for support, ideas, and encouragement (as well as the use of their computers' CPU time at night!).

At Carnegie Mellon Research Institute: The late Court Wolfe for putting together an impressive robot for inspecting aging aircraft and for encouraging me during my early years as a grad student; Chris Alberts for continuing in Court's footsteps and bringing success to the ANDI project; Chris Carroll and John Harn for help and direction in bringing vision to ANDI; and Donna Anderson and the rest of the people who contributed to bringing ANDI to life.

At the Robotics Institute & School of Computer Science: Matt Mason & Mike Erdmann for repeatedly giving me the opportunities to teach, an essential part of learning; Marce Zaragoza for helping me navigate through piles of red tape as well as lending an ear whenever I had something to expound upon or complain about (!); Rahul Sukthankar, Wes Huang, Shumeet Baluja, Justin Boyan, Rob Driskill, and many other students with whom I've had the pleasure of sharing ideas. Special thanks to Bruce Maxwell for always listening to my rantings and for helping me teach the best Computer Vision Class we could. Special thanks to Gregg Podnar and Alan Guisewite for their dedicated work with TIP and ANDI.

Finally, thanks to my advisors and committee members, Anthony Stentz, Mel Siegel, Dean Pomerleau, and Bill Kaufman for taking the time to get to know my work and for making me make it the best I could.

"All creativity is just feats of association..." ~Robert Frost, January 19, 1962

1 Abstract

In this thesis we present both a novel neural network paradigm and an approach for solving sensing and control tasks for mobile robots using this neural network paradigm. Real world tasks have driven the evolution of this methodology and its components, and we apply our methodology successfully to two robotics applications. We conclude that for some tasks, our novel modular neural network approach can achieve comparable or better performance than a traditional monolithic neural network in a much reduced training time.

We present the MAMMOTH (Modular Architecture Multi-Modality Theory) neural network paradigm, which is both an architectural blueprint and a training system for combining the internal representations of multiple neural networks each of which is trained to recognize different kinds of features. The modules in a MAMMOTH system are designed to provide *functional decomposition* of a task. That is, each module performs part of the task for a given input, and the higher levels of the MAMMOTH network combine the results to get a solution; this is different from many modular neural network techniques in which the higher level arbitrates between complete answers provided by the modules.

We apply MAMMOTH networks to several tasks, which include vision for the alignment of an aircraft inspection robot, on-road navigation, and cross-country navigation. Through these tasks we see the general applicability of MAMMOTH to real world sensing and control tasks. Ultimately, the greatest benefit of MAMMOTH is that for some tasks, low level features can be learned separately and in parallel, speeding the entire training process for a neural system, without losing any performance.

Abstract

2 Summary

In this thesis we present a novel neural network paradigm useful for solving sensing and control tasks for mobile robots. Real world tasks have driven the evolution of this technique, and we apply our technique successfully to two robotics applications. The performance of our modular neural network technique is always at least as good as more traditional neural networks, and in a certain class of problems the training time required for a best solution can be much smaller.

2.1 Tasks

Two robotics task domains are addressed in this work. The first domain is finding the rivets on the skin of a commercial airliner for alignment of the Autonomous NonDestructive Inspector of Aging Aircraft (ANDI), a robot for inspecting aging aircraft. The second domain is autonomous reactive navigation for the Navlab II, a four-wheel drive military ambulance used in on road and off-road navigation. Both of these applications involve real robots and real-world tasks, and the corresponding real-world constraints (time, uncontrolled lighting, etc.).

2.2 How we address these tasks

Mathematical modelling and neural network function approximators both have their place in robotics sensing and control tasks. Neural networks are often used in situations in which an accurate model may be very difficult to derive, but a training signal (a rich set of input/output pairs) is easily accessible. However, for any sensing task, the keys to success are the set of features the system takes advantage of in its sensing space and how the system takes advantage of them. Traditional monolithic neural networks treat all inputs equally: there is often no distinction made as to where the input came from (what sensor, what type of scenery). The neural network has to learn both the low level features and the relationships between them all at the same time.

Our neural networks are called MAMMOTH (Modular Architecture Multi-Modality Theory) neural networks. MAMMOTH networks have two parts: one part consists of a number of FeNNs (Feature Neural Networks) that are trained to recognize specific features in a sense space, the other part is the Task Net which allows the features of the FeNNs to be combined through neural learning algorithms. The use of FeNNs allows us to learn low level features in parallel, which can result in faster training times while maintaining neural network performance. Additionally, in some of our experiments, we see evidence that the decoupling of the learning of high & low level features can reduce the total number of weight adjustments needed for a neural network to learn a complex task.

For each of our two main tasks we approached the concept of modular neural networks a little bit differently. For the task of finding rivets on an aircraft skin, we first tried a model-based solution for locating rivets. Then we implemented a monolithic neural network, and finally, a modular neural network that tried to supplement the learning of the neural network with a module based on features shown to be useful in our model-based solution. For the task of driving an unmanned ground vehicle (UGV) using both color video and laser range data, we first trained a monolithic neural network. Then we trained a modular neural network which used modules that treated the features of each sensor space independently. The results for the rivet-finding did not show a particular advantage to the modular technique, while the results for the UGV task showed a tremendous savings in training time.

2.3 When is MAMMOTH applicable?

As robotics becomes more and more of a real-world endeavor, we encounter more complex tasks that are not easy to model comprehensively. We know from our own human experience that learning from example is often easier than being told how to do every step in a procedure. Yet, at the

same time, some direct instruction aids in our learning from example; breaking a problem down into simpler parts speeds up our own learning. Current artificial neural networks are considerably less sophisticated than our human brains, and we can sometimes improve the task performance of a neural network approach by using a MAMMOTH network that breaks a task down into independent subtasks. We call this *functional decomposition*. The greatest benefit of MAMMOTH is that combining independently generated low-level features saves time over learning the low level features and their relationships all at once.

Summary

3 MAMMOTH

In this chapter, we do three things. First, we describe the MAMMOTH (Modular Architecture Multi-Modal Theory) modular neural network paradigm. Secondly, we discuss a set of guidelines for when and how to apply MAMMOTH. Then we outline our general neural network techniques.

3.1 MAMMOTH

The backpropagation neural network training algorithm is a gradient descent learning technique. The surface on which we are performing the gradient descent is the error as a function of all of the trainable parameters (weights of connections). As we attack more and more complicated problems, we use more and more weights and the error surface can become more and more complicated. The *general* idea behind modular neural network techniques is that it can be easier to solve a complicated problem when it is broken down into smaller parts. By this we mean that we can either generate a mapping closer to the ideal mapping or we can generate an equally good mapping in less training time.

The motivations behind many of the concepts involved in the MAMMOTH architecture, and modular neural network research in general, have been well stated by Alex Waibel in his paper "Modular Construction of Time-Delay Neural Networks for Speech Recognition,"

> *"...It is clear that learning time increases more than linearly with task size, not to mention practical limitations such as available training data and computational capabilities. In summary, the dilemma between performance and resource limitations must be addressed if Neural Networks are to be applied to large real world tasks... Our proposed solutions are based on three observations:*
>
> *"1. Networks trained to perform a smaller task may not produce outputs that are useful for solving a more complex task, but the knowledge and internal abstractions developed in the process may indeed be valuable.*
>
> *"2. Learning complex concepts in (developmental) stages based on previously learned knowledge is a plausible model of human learning and should be applied in connectionist systems.*
>
> *"3. To increase competence, connectionist learning strategies should build on existing distributed knowledge rather than trying to undo, ignore or relearn such knowledge."* [Waibel89]

Many of modular neural network techniques described in the chapter, "Prior Work" starting on page 29, have modules which each try to solve the whole problem for a part of the input space, a philosophy which we call "Task Decomposition". For example, in [Jacobs90 and Jacobs91], each module is an expert on a particular range of speech inputs; a gating network decides which expert's outputs are used to represent the spoken vowel. Let *I* be an n-dimensional space consisting of inputs to a mapping a neural network is to learn. Let *O* be an m-dimensional space which contains possible outputs for the neural network. In Task Decomposition, each module learns a mapping from an n-dimensional subset of *I* to an m-dimensional subset of *O*. The higher levels of the modular neural network system then choose the right m-dimensional output from the m-dimensional suggested outputs of the modules.

Our philosophy is to have each of the modules learn intermediate representations, which we call "Functional Decomposition". In Functional Decomposition, for each n-dimensional point, X, in the input space *I*, each module *i* performs a mapping from k_i of the n dimensions of X into a space of intermediate representations of dimension l_i (where k_i & l_i can be different for each module). The higher levels of the modular neural network learn a mapping from the $\sum k_i$ dimensions of the union of the intermediate representations to the m-dimensional output space. One example is in our Navlab II MAMMOTH network (See "MAMMOTH" on page 92.) for steering control of our Navlab II vehicle in which one module learns to find range sensor features and another learns to find color video features, and the higher level network combines both of those results for any possible input to choose a steering direction for the vehicle (the complete input space is the union of the laser and color image spaces, while the input for the higher level is the union of the color video fea-

tures and range sensor features spaces). In short, in functional decomposition the learning of low level features is made separate from the learning of high level features.

A MAMMOTH (Modular Architecture Multi-Modal Theory) network consists of two segments, the FeNNs (Feature Neural Networks) and the Task Net. The FeNNs are trained to recognize features in their input space; the features to be learned are chosen by a human expert. Thus, there might be one or more networks in the feature level devoted to a color video sensor and each of these would be trained to recognize features in color video space, and their hidden units would be encoding these features in color video space. Other FeNNs might address features in laser rangefinder sensor space or in the input space of any modality we care to employ.

On top of this feature level is the task dependent network which unites the feature networks, the Task Net. The task level uses the encoded information from the feature level networks as inputs to a network trained to perform the whole task. These encoded feature inputs can be supplemented with additional inputs. See Figure 1, "General MAMMOTH Architecture," on page 7.

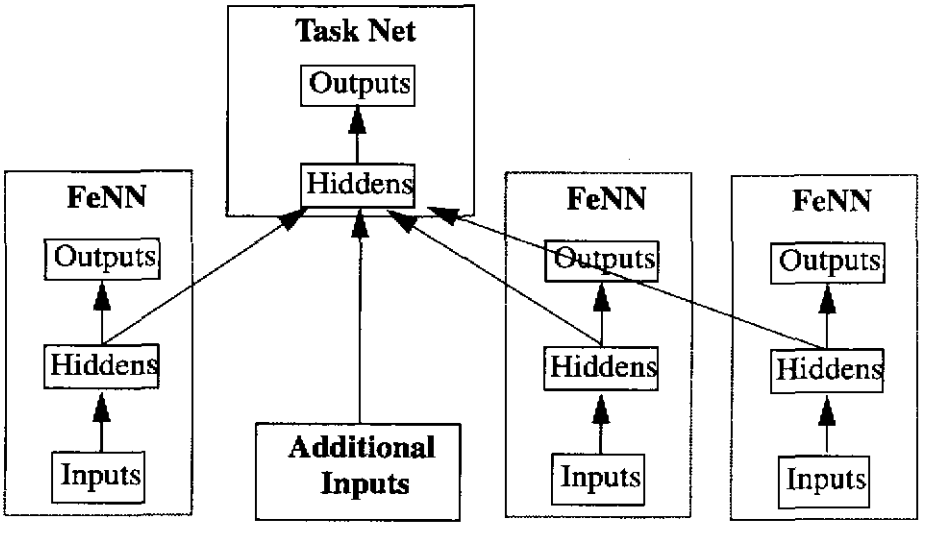


Figure 1 General MAMMOTH Architecture

3.1.1 FeNNs

The heart of MAMMOTH are the FeNNs. The goal of a FeNN is not necessarily to solve the task for some subset of the inputs, but to provide interesting features for the Task Net to work with. The FeNNs should be easy to train and should be extremely relevant. Thus, for an road navigation task, two FeNNs with potential might be a *road follower* and an *obstacle avoider*. For a rivet finding MAMMOTH network, a FeNN might be a detector of the border between rivets and the aircraft skin.

What's important in a FeNN are the internal representations generated and not the outputs. The output signal of a FeNN could be irrelevant or even misleading. For example, in one of our tasks to be described later, one of our FeNNs is used in a system that *seeks* or *ignores* vegetation with the Navlab II, but the FeNN is trained to *avoid* vegetation since that was an *easier training signal to generate*. We are interested in low-level features. We can think of the FeNNs as virtual sensors that find significant low-level features.

3.1.1.1 Training

The FeNNs used in this work are feed-forward multi-layer perceptrons trained with the backpropagation training algorithm [Rumelhart86], although they could be trained with other neural network training algorithms (features generated through models would be classified as part of the "Additional Inputs" part of the MAMMOTH paradigm). Conceptually, the FeNNs are trained independently before the Task Net, and the weights of the connections are frozen. We freeze the weights so that the features do not mutate away from what we believe are important features.

3.1.2 Task Network

The Task Net is also a feed-forward multi-layer perceptron, whose inputs are the harvested features from the FeNNs (as well as whatever additional inputs the user provides). The outputs of the Task Net are the outputs of the ideal mapping of the task. For example, in a road navigation task, the inputs might be the internal features generated in road following and obstacle avoiding FeNNs and the output would be the desired steering direction for the complete road navigation task.

3.1.2.1 Training

After the FeNNs are trained and their weights are frozen, a set of task exemplars (input and desired output pairs) are presented to the MAMMOTH net, and the Task Net is trained. Activation feeds forward through the FeNNs to the Task Net, although in the learning stage, the error propagates

only to the inputs of the Task Net (the FeNNs are frozen). The task exemplars (training examples) are the same exemplars we would use for a monolithic network to try to learn the same mapping from inputs to outputs.

In practice, though, once we have trained the FeNNs, we can present the inputs from the Task Net's training set to each FeNN. We can record the activations of the hidden units of the FeNNs, and construct a training set for the Task Net that consists of these hidden activations as inputs along with the corresponding outputs from the Task Net's training set. Then, training the Task Net is much faster since the activation through the FeNNs does not have to be recomputed at each stage of the training. For example, a particular FeNN could have 1024 input units, and 7 hidden units, and we save over 7000 activation propagations if we use this method. Note that when using the MAMMOTH network to perform a task (as opposed to learning a task), we still need to perform the activation propagation straight from the input units.

3.1.3 *The benefit of MAMMOTH*

The motivation for MAMMOTH is that we can frequently provide examples of useful low-level features more easily than we can model them. In this sense the MAMMOTH paradigm is a convenient expression for transforming or preprocessing data with neural networks. In turn, this implies that we don't have to try to solve a task monolithically when we use neural networks (just going from raw inputs to raw outputs). Neural networks are useful for both low and high level mappings for sensing and control tasks.

However, by expressing MAMMOTH as a modular neural network instead of just neural preprocessing, we gain two things. First, we allow ourselves to see that the internal representations of a neural network can be more useful for modularity than the outputs in many cases, an idea supported by Waibel [Waibel89]. This allows us to use training signals that may seem nonintuitive at first, but which generate the right features most easily (such as the network that seeks or ignores vegetation by using a FeNN that avoids vegetation).

A simple example of how using the hidden units as the input to the Task Net can help is follows: We can easily train one FeNN to avoid big obstacles in range sensor space, and another FeNN to avoid large grass mounds in color camera space. If the vehicle comes up to a single large grass mound on the right side of both images (range and color), each module might say "turn left to avoid the only mound visible", and we would want the Task Net to also say, "turn left." However, if the vehicle comes up to two mounds equally far away, one of which is a small dirt mound and the other of which is a large grass mound slightly to the right of the dirt mound, the outputs of the FeNNs could give us the same signals: the range FeNN could say "avoid the larger range obstacle, so turn left", and the color FeNN could say, "turn left to avoid the only thing I see, a big vegetation mound". In fact, for our task we may prefer to run over the vegetation mound, but there is no way

using just the outputs of the FeNNs to distinguish this case from one in which the only obstacle is to right and both FeNNs say turn left. For an illustration of the two mound situation, see Figure 2 on page 10. Using the lower level features (hidden unit activations) of the FeNNs would let the Task Net decide to turn left in the first case, and right in the second.

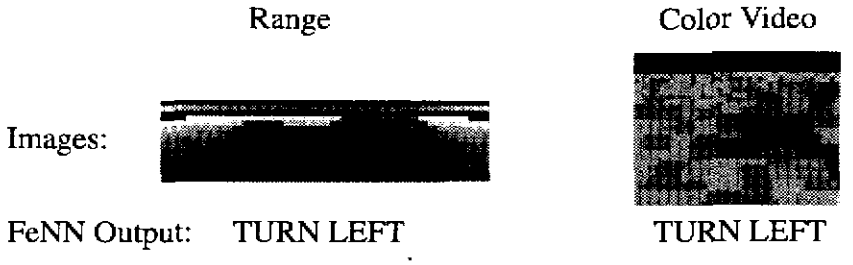


Figure 2 Why the outputs of FeNNs are not best inputs to Task Net (in video image, the dark patch on the right is vegetation, while the dirt mound on the left cannot be distinguished from background)

Secondly, in the modular neural network paradigm, MAMMOTH's applicability to sensor fusion tasks becomes plain. The FeNNs are well suited to learn features specific to sensor modalities and the Task Net performs the sensor fusion only on the useful features.

3.2 *Guidelines on when and how to apply MAMMOTH*

We are concerned with a particular class of problems, which we can loosely define as hard-to-model and easy-to-train (the latter meaning that training signals are available). Neural network techniques, and MAMMOTH in particular, are applicable to these problems, and it is our goal to present guidelines which will help us apply our neural networks and MAMMOTH to this class of problems.

Some robotics sensing and control problems can be solved with mathematical models, whether they be geometrical, topological, differential, or other precise techniques. By thinking of each problem as a mapping to be determined from an input space to an output space (as per our general prin-

ciples), this means that we can come up with a relatively small number of transformations, algorithms, or equations that approximate the ideal mapping fairly closely - or even match it exactly (though this is rare in the real world). A simple example might be finding the center of a white target on a black background in a video image. A more complex example might be transforming a video image of a road into an image of edges, joining the edges into curves, computing the curvature of the road from the curves, and calculating what steering direction would correspond to following that curve.

However, there is a large class of problems in robotics and the real world in general, which are hard-to-model. In fact, both of the examples in the above paragraph can be made hard-to-model when put into the appropriate context. For instance, if we want to compute where the center of the white target is in the real world, we will have to know something about the camera parameters, and some of the best camera calibration techniques rely on iterative function approximation [Tsai86 & Tsai88]. Furthermore, road following based on edge detection is often more complicated than it first seemed to researchers, and one of the most successful road followers uses a neural network function approximator to map directly from the video image to the steering direction [Pomerleau92].

In order to use a neural function approximator, or any other function approximator, what we need is a well defined training signal (inputs and corresponding outputs) and a way to gather examples of this training signal, called exemplars. For camera calibration, we put a known calibration pattern in front of the camera; the inputs are the points in the image, and the outputs are the "ground truth" from the calibration pattern. For road following, the training signal consists of video images and the corresponding steering directions gathered when a human is driving over a stretch of road.

The MAMMOTH paradigm lets us decompose the mapping from input space to output space. We use neural networks both to find useful features and to solve the whole Task with these features. The questions are how do we know when to apply MAMMOTH and how do we use the MAMMOTH paradigm?

First, we need to identify a problem as hard-to-model and easy-to-train (as defined earlier). Next, it is important to remember that MAMMOTH, as presented, does not automatically perform functional decomposition. We need some task knowledge about useful decompositions of the mapping from input to output space. Furthermore, a task for which MAMMOTH provides the greatest benefit is one in which part of the mapping needs to be relearned frequently (since MAMMOTH's main benefit is a reduced training time, frequent retraining results in the largest savings).

One good way to come up with a decomposition is to create a module which duplicates the performance of a model that solves part of the problem (in our rivet-finding task, we know edge detection solves part of the problem, so we use a module that performs edge detection). Another good decomposition is decomposition into features in different sensor spaces which can be combined to come up with task level features (for our Navlab II tasks, we train one module in range image space

and one in color image space). However, it is important to remember that we still need to be able to get reliable training signals with some ease. As one of the benefits of MAMMOTH is that training time is shorter to achieve a certain level of performance, we don't want to negate this by having to spend too much time generating training signals.

A summary is shown below:

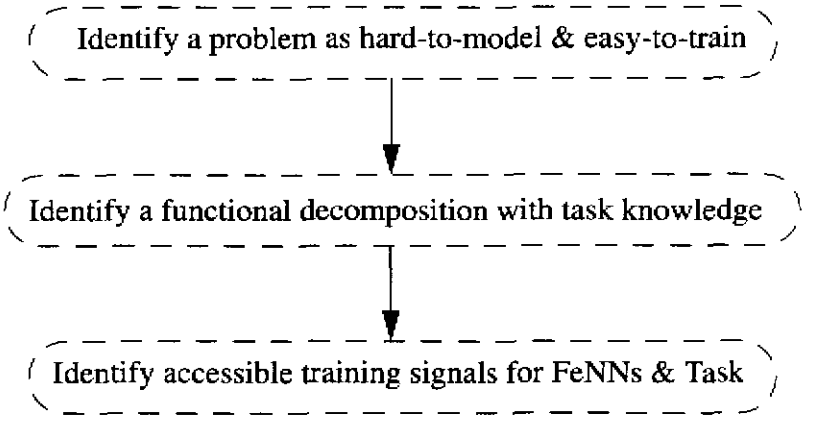


Figure 3 How to use MAMMOTH

3.3 General Neural Techniques

The neural networks we use are feed-forward multi-layer perceptrons trained with backpropagation. These networks have at least one input group, at least one layer of hidden units, and a layer of output units. Each unit (also called a node or neuron) in the network has a certain level of activation and propagates that activation to the next layer through weighted connections to some of the nodes in the next layer. Learning occurs by adjusting the weights of these connections (for an introduction to neural networks, see [Wasserman89] or [Caudill90]).

3.3.1 *Architecture*

Two of the most important issues in using neural networks are the network architecture and the construction of the training set. The issues in choosing a network architecture are how many nodes are in each layer of inputs, hiddens, and outputs, and the pattern of connections going from one layer to the next. In this work, the answer to the latter question is that all monolithic networks, FeNNs, and Task Nets are fully connected. This means that each node connects to every node in the next layer. This is because within the bound of the functional decomposition we decide on, we let the neural networks determine the important features based on all inputs. There is no reason MAMMOTH networks have to be fully connected, but we have not explored ones that are not.

The numbers of inputs and outputs are dependent on the goals of the task. The number of hidden units roughly corresponds to the computational ability of the network; you want as many as you need to closely approximate the ideal function, but not many more. Too many hidden units can result in improper interpolation and extrapolation, which means poor generalization.

Since choosing the right number of hidden units is of importance, there has been great interest in methods for determining the right number for a given task and data set. Analytical approaches have been proposed for determining both the complexity of learning a given mapping (which helps determine the size of the training set), and the complexity of representing a given mapping (which helps determine the number of hidden units). [Haussler89] describes a complexity theory for neural networks based on the Vapnik-Chervonekis (VC) dimension, and [Cybenko89] outlines an approach for the design of multi-layer perceptrons based on a related concept involving the "algebra of sets". The analytical techniques tend to be based on the concept of the size of a minimum covering (the details of which vary with the techniques) of the input and output sets of a mapping.

One problem with the analytical techniques is that it is not always clear how to apply them to large real world data sets. Another issue is that our data sets change frequently. In fact, we shall see that MAMMOTH's greatest benefit (reduced training time) arises when there is a *frequent* need to retrain on new data. Thus, a new analysis would need to be performed before each retraining. For these reasons, a more common approach to choosing the number of hidden units for a network is to perform some a search on a limited space of possible architectures. Real world real-time robotics tasks constrain the size of a network anyway, so we can perform an exhaustive search on a small number of different architectures[1]. This is what we do (see the description of ANGEL below). Other types of searches (beyond an exhaustive one) are also useful. One interesting area is the use

1. One of our monolithic networks for the Navlab II has six hidden units and we can do a forward pass of that network in 60 milliseconds with a current workstation. Sixteen hidden units would make this network take 180 milliseconds. We wish to keep the neural computation to under 250 milliseconds if possible so that the whole system can cycle above 1 Hz.

of a genetic search of the space of potential architectures [Kuscu94]. This area holds promise because we can expand our search space to include not just the number of hidden units at the first hidden layer, but also the number of layers, the pattern of connectivity, and learning parameters.

One more area of research involves training a single network with many hidden units and pruning the hidden units to remove those which are not being heavily utilized in the mapping [LIM93] and [Karnin89]. This can almost be thought of as the opposite of growing networks with a technique such as Cascade Correlation (see "Cascade Correlation" on page 37). The advantage of a pruning technique over a search is that it can be faster than training a large number of networks in the search. The disadvantages are that it is not as straightforward to implement, and there is a risk of over-pruning and losing valuable learned data.

For our purposes, we have enough constraints on our tasks such as the need for fast execution time, the need to retrain networks often, and a history of related tasks to influence the design of our networks [Pomerleau91], that a simple exhaustive search of a limited space of potential architectures will suffice. Furthermore, as there may in the future be a need to expand the search space to include other parameters (such as those mentioned above), we still need to find network architectures and paradigms such as MAMMOTH which reduce the training time in weight adjustments for a network.

3.3.2 *Training Set Construction*

The training set construction is also largely task dependent. Typically, the more examples you can provide of the ideal function, the better (though at the expense of training time), as there is noise in the data for our tasks, and more examples help smooth out errors. Size of the training set is only limited in practical applications by the amount of time it takes to gather it and the amount of memory available for storing examples (when they can't be gathered on-the-fly). Theoretically, the main issue is that the training set should be balanced between all parts of the function which are important. Too many examples from one region of the input space and not enough from another part of the input space will result in one part of the function being learned well to the exclusion of the other since there is a finite amount of computational capacity in a given network.

3.3.3 *Completing Training*

For all of our research, we use a simple approach for deciding when to stop training a particular network, whether it be a FeNN, a Task Net, or a monolithic network. Training is done in "batch" training, which means we present our whole training set to the network and accumulate the error over the whole set, and then perform the weight adjustments just once for each presentation of the complete training set. We train the network with a fixed learning rate and momentum term[1] (exact

values are empirically determined for each task, but the learning rate tends to be low and the momentum high). When the summed error of each of the nodes of the output layer on a test set (the same size as the training set, but with distinct exemplars) stops improving (or begins to increase) for an extended period of time, we declare the network to be trained and then we use the weights of the network saved just after the error reached its lower bound plateau.

In our earlier work, we trained a number of different networks with different numbers of hidden units one at a time and compared their performance on the test sets to choose the number of hidden units that would let the network learn the function best. This was fairly time consuming and it meant we usually limited the scope of our search to a relatively small number of hidden units. The range was usually determined by comparing our estimate of the problem complexity to known solutions; for example, from [Pomerleau91] we know that four hidden units is sufficient for road following, so our early road following FeNNs used around four hidden units. We believed obstacle avoidance to be harder, so we tried a number of different networks with a few more hidden units (the final numbers used are listed with each experiment in the chapters "ANDI" & "Navlab II").

Later in the research, we automated the process for training networks and used a system we wrote and developed called ANGEL (Automatic Network Generator and EvaLuator). Angel (described in the Tools Appendix) would train many networks in parallel on many different available machines. With ANGEL we could repeatedly search through a larger number of network architectures much more quickly. The output of ANGEL was a series of trained networks with weights saved at regular intervals through the training process, as well as measures of each network's performance at different stages in training. These measures could be examined and graphed to find the best network and training time.

1. Since backpropagation is a gradient descent technique, the learning rate is the magnitude of the step along the "downhill" direction, and the momentum term is added as an attempt to keep the direction of the descent stable amidst small perturbations in the error surface. Approximate values for many of our experiments were a learning rate of 0.002, and a momentum term of 0.5.

An example of the error for a number of hidden units over a number of different training epochs is shown below in, where 12 networks with the same number of inputs and outputs but different numbers of hidden units (1-12) were trained for a range FeNN.

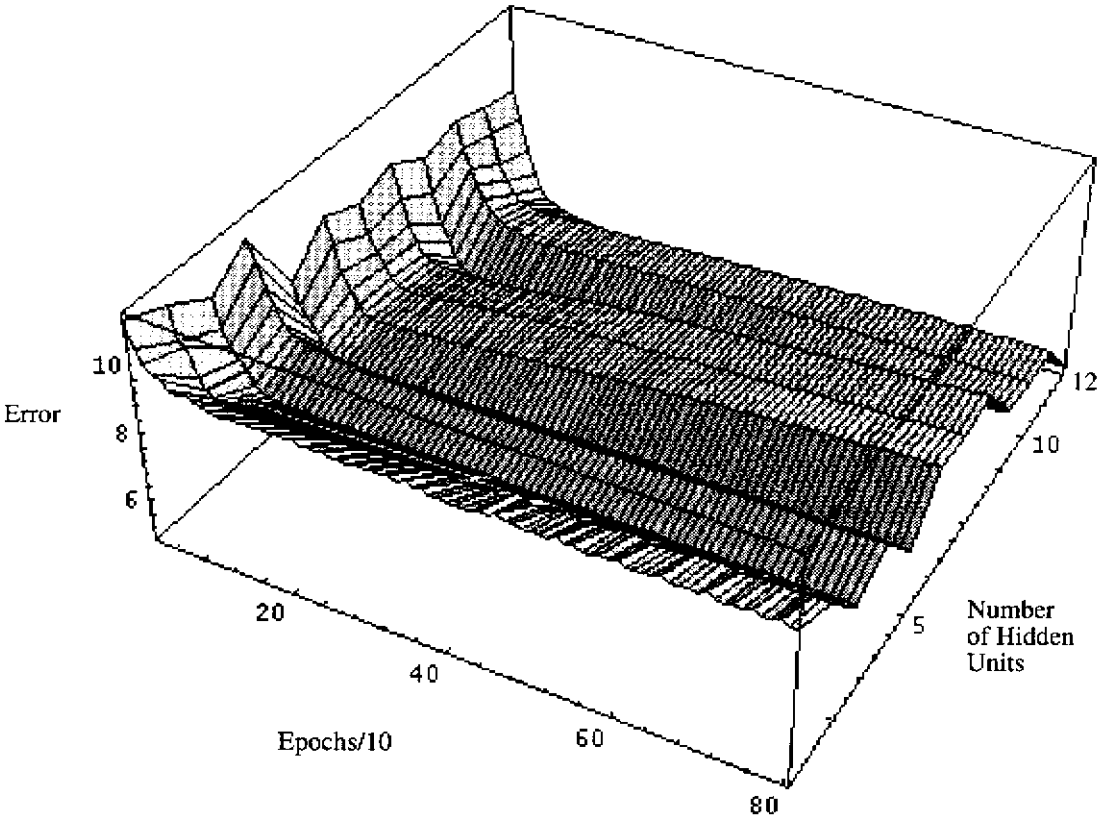


Figure 4 Output of ANGEL for a range FeNN. For this graph, each network was trained for 800 epochs. The vertical axis is relative summed error for the network's output units. The axis marked 20 to 80 is epochs in increments of 10 (so, 80 corresponds to 800 epochs). The third axis shows the number of hidden units. In this example, the two best performing networks for the test set had 9 hidden units (at 141 epochs) and 12 hidden units (at 212 epochs). Repeat training (with different random weights) showed the 9 hidden unit network to be consistently good.

As expected, the network's error on an *independent test set* reaches a low point after some training, and then begins to rise again, although the performance on the *training set* may continue to improve with more training. It is important to look at the performance on the independent test sets

so as to avoid overlearning the training set to the exclusion of performance on additional input examples.

3.3.4 Input Scope

For sensing and control problems like those we examine, one of the most important choices to make is the scope of the ideal function we want to learn. All of the tasks we examine involve mapping from sensor data in some form, although MAMMOTH's usefulness is not necessarily limited to raw sensing mappings. For our purposes we will refer to the raw sensor data as the sensor image regardless of the nature of the sensor (CCD camera, laser range sensor, or even sonar data would be called a sensor image).

In some cases, we want to make a mapping from an entire sensor image, and in some cases we want to merely transform small parts of the image. The scope of the architectures we use for image processing here can be broken down into two general schemes, image-local and image-global. Image-local architectures look at just a small part of an input image and image-global schemes look at the whole image at once (even if it has to be reduced in resolution for computational reasons).

3.3.4.1 Neural Operator Architecture

3.3.4.2 Our image-local network architecture is called the operator-architecture. An operator-architecture network functions just like a typical computer vision operator such as an edge detector or a convolution operator. An operator-architecture network consists of an input layer, called the retina, which is typically filled in with the intensity values of the image in a small neighborhood around a given pixel. The output is usually a continuous classification. By scanning an operator-architecture network over an entire image, we can generate a new image from the outputs of the network. Our operator-architecture networks are fully connected from their retina to their hidden units to their output. An example is shown below.

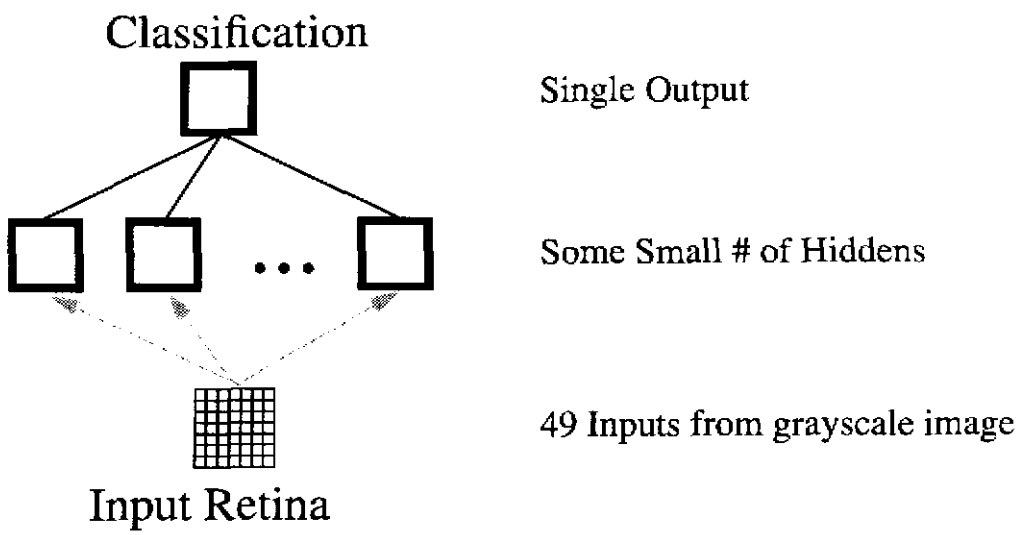


Figure 5 Operator Architecture Network. This network gets scanned over entire image.

In choosing to use an operator-architecture network, we need to know *generally* what sort of features we are looking for, although the network will develop its own representations. For an operator-architecture network, the size of the retina is crucially important. Since we are trying to find a particular feature, the operator's retina should be large enough to contain an example of that feature. If the retina is too big, we can introduce incidental features into the retina which can interfere both with learning and classification. If the retina is too small, we simply cannot see the features we want to find. Of course, the input retina can be as small as a single pixel if the ideal mapping can be made from the intensity values of a single pixel. An example is the rivet finding operator-architecture networks described later in which the retina's size is chosen to be on the order of the size of a rivet in the images.

3.3.4.3 ALVINN Style BP

Our image-global network architecture is based on the ALVINN image processing network [Pomerleau91]. The input layer of one of our image-global networks is a complete image from a given sensor. For our purposes, the output is usually a steering command corresponding to the input

image. A sample of an ALVINN-style network is shown below (with a range image as input); note that each input node is connected to each hidden node and each hidden node is connected to each output node.

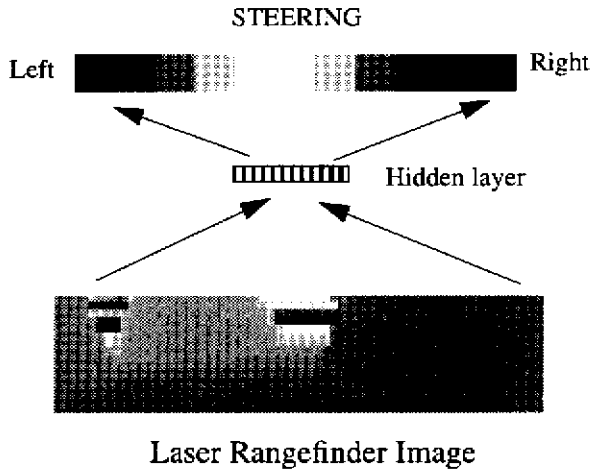


Figure 6 A sample image-global network.

For practical reasons, we usually cannot have the input to an image-global network be a complete high-resolution image. The time required to process a full high-resolution image-global network would be prohibitive for most of our real-time navigation tasks. But on the other hand, if we decimate the input images too much, we can be operating at a resolution at which the network cannot see significant features. Thus the resolution of the input image is of comparable importance to the retinal size for an operator-architecture network. For our Navlab II tasks, we made sure the large mounds were clearly visible in the reduced resolution images we used as inputs to our networks (see Figure 2 on page 10 for examples of the images).

4 *Background*

This work is distinctly a robotics thesis, with motivations, implementation issues, and implications that, when taken as a whole, are unique to the field of robotics. The tasks we address in this thesis were chosen because they presented problems which needed to be solved, and solving these problems was our motivation for the work presented here.

Robotics research encompasses many fields. Robotics in general has much in common with computer science, electrical engineering, mechanical engineering, and physics. In particular, sensing and control for mobile robotics overlaps computer sensing (including computer vision), sensor fusion, and machine intelligence (including neural networks). Robotics is often motivated by the single goal of "Make the robot work in the real world!" For example, where computer vision as a stand alone field may be concerned with finding all findable features in an image, robotics is concerned with finding enough features in an image to be able to safely move the robot - and doing it with as few resources as possible.

In this chapter, we define the scope and focus of this thesis and its relationship to some of the fields that make up robotics. We explain the issues that went into our decision making and system design.

4.1 What sort of thesis is this?

This thesis is primarily a robotics thesis. Robotics has often been treated either as the intersection or sum of several other fields, or as a label to conveniently apply to one of its component or contributing fields. Robotics does draw on other fields. We address issues of machine intelligence, sensing, and control (among others). We apply neural network solutions. But, ultimately, this thesis is about robotics and an approach to engineering a solution for two specific robotics problems.

Engineering a robotics system involves constraints on time, computing, and applicable techniques, and demands a particular heartiness (robustness) in the systems. For a continuously moving robot or for a robot that is to outperform a human in a particular task, computational speed is critical. For a self-contained mobile robot, power and weight restrictions limit the available computing resources. For a robot in an unstructured environment, techniques that are based on high level abstractions may be difficult to implement. And with these constraints, roboticists must make systems that perform nearly flawlessly, because in addition to other motivations, the bodily integrity of the robot is frequently at stake.

This unique perspective found in robotics can be summed up by extending Brooks' preconditions for machine intelligence [Brooks89]. He claimed that real intelligence requires that the intelligence be situated and embodied. For robotics this means that the robot can sense and can act in its environment. Moreover, in our tasks, the robot always has a specific goal. Consequently, the circumstances of robotics that drives a system's development are that the robot is always situated, embodied, and motivated.

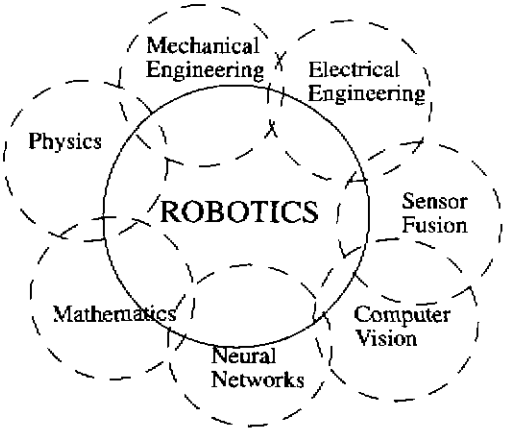


Figure 7 Robotics is not strictly the sum or intersection of neural networks, computer vision, sensor fusion, mechanical engineering, electrical engineering, physics, mathematics, etc.

This work is a study of two situated, embodied, and motivated robots, and a study of robotics. Both the individual components and the whole are important. In order to understand this work as a robotics thesis, it is important to see what questions we ask about our particular work's main components: computer vision, sensor fusion, & neural networks.

4.1.1 Computer Vision Issues

Perception is the first critical component in our robotics systems, and computer vision is one of the first areas to which we feel we can make a contribution. As computer vision has been studied seriously since the late 1960s, there is already a large library of techniques we can draw from. There are operators (functions mapping from pixel space to feature space) for extracting features, classification techniques for pixels, and methods for manipulating image regions. We have complicated real-world scenes and the speed and computing limitations of working with mobile robots. The constraints of our tasks force us to examine several interesting computer vision problems.

4.1.1.1 Image-global feature extraction vs. Image-local feature extraction

One of the issues that we have to cope with is the question of how to extract high level features from images, which are essentially just a collection of pixels. The traditional Computer Vision model involves finding low-level features in the images and then finding high level features from

the low-level features. These low-level feature detectors are what we call "image-local" and their inputs are a pixel and possibly a few neighboring pixels; using image-local techniques makes the low-level image processing fairly expensive for serial computers. One of the robotics tasks involved in this thesis allows the use of local operators, primarily because the higher level features are simple (see "Rivet Finding Methods" on page 55). On the other hand, if the higher level features are more complicated or can be represented by large areas of image space, we may need to use an image-global technique which extracts high level features directly from all of the pixels in the image. Our first computer vision question is "Should we spend most of the robot's computing resources on image-local or image-global techniques?"

4.1.1.2 Resolution issues

Real-time robotics forces us to be aware of computational expenses when extracting either global features or local features. Finding every feature at every resolution is computationally expensive. Even when we're not sure how to model a feature, we still need to know what size our important features are. We frequently have to use lower-resolution versions of our raw images to process them quickly enough. Our second computer vision question is "What resolution gives us the greatest benefit in terms of finding important features at the least processing cost?"

4.1.2 Sensor Fusion Issues

A major difference between computer vision as a stand-alone discipline and computer perception for robotics is that one sensor modality is frequently insufficient to accomplish a complex field task. Sensor fusion thus becomes important to solving our robotics tasks. The constraints of our real-time robotics tasks make us explore several different areas of sensor fusion.

4.1.2.1 Efficient sensor fusion

Individual sensor calibration and multiple sensor registration can involve complex and time consuming geometric transformations. Much of this registration and calibration computation performed is irrelevant to the task at hand. In our systems, the sensor fusion is driven by the tasks. When we solve a sensor fusion problem, one way we can speed things up is by performing the fusion not on the raw pixel data, but on the features useful to the task. The question to ask is, "On what features should we base our fusion to still get good results but not to waste processing time?" Our modular neural network techniques gives us a good mechanism for generating the features and fusing them - once we know which features are useful.

4.1.2.2 Virtual Sensors

The final area of sensor fusion which is of interest to us is the use of Virtual Sensors [Jochem96]. A broad definition of Virtual Sensors is an abstraction of a sensor that performs some transformation on the raw data of a sense space or which extracts higher level features from raw sense data. In this way, multiple "virtual" sensors can be made from the same sense modality for integration into the task execution. For example, if the only physical sensor is a CCD video camera, we might create Virtual Sensors that extract edges or circles from the image. The Feature Neural Networks (FeNNs) of our modular neural network approach can be thought of as virtual sensors. The question we ask in a problem is "What features might be good candidates for being the subject of a Virtual Sensor?" These Virtual Sensors can be thought of as neural preprocessing.

4.1.3 Neural Network Issues

The primary method we developed for solving our robotics problem is a neural network technique. This thesis represents some promising initial experiments with a particular modular neural network paradigm as well as presenting results obtained with monolithic neural networks. There are a number of neural network issues and questions which we examine in each task we approach.

For our purposes, "monolithic" neural networks are those in which all of the units of one layer of the neural network connect to all of the units at the next layer down and that the entire network is trained simultaneously (all weights are adjusted in each iteration of the training algorithm). For complex problems, this may not scale well since the training time increases more than linearly with the size of the training sets and network (see [Waibel89] for some discussion). Although the artificial neural networks used today (including the ones in this work) are of a very small scale compared to the biological brains of humans and animals, the consensus is that modularity is the eventual direction (see [Jacobs90], [Smieja92], [Hampshire89], [Waibel89], and other works in "Prior Work" on page 29). Although the particular design of our MAMMOTH networks will be discussed later, we can now describe several of our modular neural network issues.

4.1.3.1 How can we learn features?

Central to any neural network research is the idea that it is often difficult to formulate a mathematical model to solve a problem. Thus, we learn from examples. We must ask, "How can we get good examples of our features?" Constructing the example sets is often the central aspect of learning visual features. Our training sets are extracted from a training signal which consists of inputs and outputs for each network.

4.1.3.2 Module architecture & training

One of the central issues in modular neural networks is the architecture and goal of the modules. Frequently, the modules are identical in architecture [Hampshire89]. Often they work on the same input and output spaces and each solves the same problem for different sets of vectors in the input space (and the higher layers of the network arbitrate between the modules to choose *the right one*). But lower level modules don't have to be identical. For fusing different sensing modalities, each modality typically suggests its own architecture for a module due to the unique shape, size, and resolution of the sensor. So, for each task we want to ask, "What should the modules look like, and how should we train them?"

4.1.3.3 Functional Decomposition & Integration of A Priori Knowledge

Our motivation for using modular neural networks is that if we can train a part of a network to find some feature that we know is important in solving a problem, then the rest of the problem solving should be easier (meaning we can learn our desired mapping faster or better). Breaking a problem into smaller subproblems, each of which contributes to the combined solution is called *functional decomposition*. This leads to the question, "How can we turn our a priori knowledge of the task into a functional decomposition?"

4.1.4 Robotics Issues

Ultimately, the robotics questions we answer are the questions related to the individual tasks. Each task has a different set of sensor inputs and desired outputs, and the following sections describes what broad questions we intend to address with our specific tasks (which will be detailed in the next chapter). More generally, though, we have helped show the applicability of neural network techniques and modular neural network techniques to real world robotics tasks. We believe the functional decomposition approach of our neural networks can be applied to a number of real-world sensing tasks.

4.1.4.1 Rivet Finding: Feature Extraction in Extreme Noise

Our first task is finding the rivets on an aircraft skin. This is feature extraction in a very complex and noisy environment. Feature extraction is both feature finding and feature filtering. In feature finding, we must ask the question, "What subfeatures are common to all of our features (rivets)?" For feature filtering, we must ask, "How can we decide when our subfeatures constitute the sought after feature?"

4.1.4.2 Navigation of a wheeled vehicle using range and color data

We address both Road Navigation and Cross Country Navigation in this work. Our systems integrate the sensor data from both sensor spaces with neural networks which allows the learning of non-linear fusion.Road following is a fairly widely studied problem with several robust solutions [Pomerleau91]. Road navigation is road following with obstacle avoidance, and has its own set of issues. Clearly, the main questions are, "Can we, and, if so, *how* can we modify an existing road following method to perform road navigation?"

Finally, most cross country navigation scenarios involve vegetation, despite the fact that most cross country navigation systems to date have no way to sense color data. Thus, the first question is, "What sensing modalities are most appropriate for cross country navigation in the presence of vegetation?" The second question is, "Which features are most useful in these sensing modalities?", and the third question is, "How can we find these features?"

4.2 Goals

This thesis evolves an approach to solving robotics perception-navigation problems. There are problems which can be well solved by neural network techniques, because they are difficult to model and a reliable training signal is available. The MAMMOTH paradigm gives us a tool that allows us to solve some neural learning problems much more quickly than we could with a traditional monolithic neural network.

5 Prior Work

To explain the derivation and significance of this thesis, we must examine prior work done in several different areas. The first areas place my work in the context of other approaches to similar tasks. The latter areas place my work in relation to prior work in neural networks for navigation and in relation to prior modular neural network approaches applied to different problems.

5.1 Sensor Fusion

Sensor fusion is a broad term which includes many different techniques and many different types of sensors. We briefly describe in the section several different sensor fusion systems in order to put our system into context. For each system, we outline the basic techniques and the types of sensors. For reference, our system for the Navlab II fuses color video data nad laser range data using neural networks. The fusion is not performed on the raw data or fully interpreted data, but on intermediate representations learned by the neural modules. Sensor fusion is also performed in the section "Carbot" on page 35 for a task that is closely related to our tasks as it is a reactive system for a mobile robot.

5.1.1 *Fusion on Multiple Geometric Observations From Different Sensors*

One common area of research is research into using multiple sensors to generate a geometric map of the environment for the purpose of robot localization or navigation. Three examples are presented below. These stand in contrast to our system for several reasons (described below). Primarily, the task of precise localization requires precision maps (whereas our system is strictly reactive), and these systems require detailed and precise modelling of the sensors (while our system is rapidly reconfigurable for new sensors, resolutions, or mountings).

5.1.1.1 Hager

A rich area of the field of sensor fusion is the fusion of data from different sensors, each of which produces some sort of geometric model of the environment. Hager [Hager90] describes a system which includes detailed modelling of sensors and a grid based probabilities density method for mapping the environment with multiple sensors. Although he talks about and examines a hypothetical system using multiple cameras as well as sonar, the implementation is limited to a camera only system (although the camera can move). Our system is different in that we do not require detailed modelling of the sensors. This makes our system more flexible for rapidly changing the sensor configuration. Hager's system does, however, produce usable and reusable maps of the environment.

5.1.1.2 Durrant-Whyte

In a similar vein, Durrant-Whyte uses Bayesian probabilities in computing maps of the environment from multiple, well modelled sensors [Durrant-Whyte88]. The sensors he used were stereo video cameras and tactile sensors mounted on a robot arm to build a probabilistic geometric representation of the environment. As with Hager, this approach has the advantage of generating explicit and detailed maps. The disadvantage, though, is that it does not allow the rapid flexibility of a system like ours. Furthermore, the environment we study is less controlled.

5.1.1.3 Courtney

A more recent example of a similar approach can be found in Courtney and Jain [Courtney95]. They use three sensor modalities, ultrasonic range sensors, video cameras (providing lateral motion vision), and infrared proximity detectors for generating grid maps of a hallway-type lab environment. Each sensor generates its own grid map and sensor fusion and robot localization is performed upon features found in each map. Although the sensor suite is not dissimilar to ours, one major difference is that the camera data is used for geometric feature detection just like the other ranging sensors. One commonality is that they perform the fusion based on an intermediate representation (their invariant features). But the large differences in technique and goal remains: we use neural

networks to perform fusion on neurally generated features (predetermined only through the training of the FeNNs), and we want to find features that exist in different sensor spaces that don't necessarily simply map to generic geometric obstacles. As above, though, Courtney and Jain's system does generate lasting maps of the environment.

5.1.2 *Self-Organizing Maps*

Tabakman and Exman [Tabakman93] detail a system that uses a connectionist approach to fuse data from multiple sensors. Although it is not clear what the long term goal is, this system which can be used for localization using range and angle data (presumably from a suite of similar sensors such as sonars) touts several advantages which we also claim. Note that the self organizing maps involved are not overly similar to the perceptrons trained with backpropagation which we use; in fact, as they are self organizing, the concept of training signal is no longer defined by input/output pair, but simply by an input stream. They call their system a multi-layer self-organizing feature-map system (ML SOFM). One of its large advantages is that due to the multi-layer nature of it (modular), much of the processing can be done in parallel. This is a claim we make for our system, but we take it a step farther by claiming that by performing our functional decomposition we can decouple learning high and low level mappings and save time beyond simple parallel processing of modules.

5.1.3 *Navlab I*

As early as 1986, researchers on the Navlab I autonomous van at CMU were concerned with fusing the data of video cameras and laser range finders. In an early tech report proposing a sensor fusion system, Shafer wrote, "*...sensor fusion will take place on fairly completely interpreted data rather than on raw or intermediate sensor data*" [Shafer86]. One reason for this is that raw sensor data generated by different cameras may be difficult or expensive to register explicitly due to calibration and timing issues (images taken at different times). In our work, we fuse partially interpreted data in the hidden units of our FeNNs. Our fusion method is also much different in that it is neurally based. Our motivation for fusing intermediate data is that fusing raw data can be too expensive, and fusion fully interpreted data can mean we've filtered out too much information before the fusion.

5.1.4 *Modelling Rugged Terrain By Mobile Robots With Multiple Sensors*

Kweon's Ph.D. thesis from Carnegie Mellon University [Kweon91] includes an example of sensor fusion using the same sensors which we use in this thesis: a laser range camera, and a color video camera (in fact, she used the very same laser range camera that we use). The purpose of Kweon's work was generating terrain models or maps using the various sensors (much of the thesis was con-

cerned with just the range data). The technique used is simple calibration and registration of pixels from one sensor to the other, and the result is a map whose grid points contain an elevation and a color. Our technique fuses higher level data representations, and is control-task-dependent, meaning we fuse the data directly in computing a steering direction for our robot. Our method has an advantage in efficiency, but Kweon's method generates a permanent map.

5.2 Cross Country Navigation

Since the late 1970s there has been serious research in off-road navigation from the military and space exploration sectors. We are concerned primarily with low-level navigation systems, and especially with reactive systems. Perhaps the biggest difference between previous cross country navigation systems and our system is that the previous systems *did not address the issue of vegetation*. With that in mind, here are several of the key systems and their highlights.

5.2.1 Hughes ALV

The earliest autonomous off-road navigation system was developed at Hughes for their Autonomous Land Vehicle (ALV). This system built a model of the world through external sensors (the ERIM rangefinder), and internal sensors which gave readings of vehicle roll, pitch, etc. This system modelled kinematics only, was low speed, and was not extensively tested. Any dangerous vehicle configuration was considered an obstacle [Daily88]. In our system, we fuse color video data with the laser range data to provide vegetation handling capabilities.

5.2.2 Robby

The Jet Propulsion Laboratory (JPL) developed Robby for cross country navigation. Robby sensed with stereo and navigated using models whose input was the results of the stereo perception system. Robby achieved best results of 4cm/s. Robby's focus was building terrain maps from the stereo data. Our approach to perception differs greatly from the Robby system in that we use data from different sensors and fuse the data in a task dependent manner.

5.2.3 *Level I FGN*

Three full off-road navigation systems have been or are being developed at CMU for the Navlab II, a modified army HMMWV (High Mobility Multi-Wheeled Vehicle, a 4-wheel drive ambulance). Each of them builds an explicit mathematical model of the robot's environment. The first one was the Level I Full Geometry Navigator (FGN) [Brumitt92]. This systems generates a map of the terrain from laser range data and generates trajectories by simulating moving the vehicle through the generated map. The FGN system modelled dynamic constraints as well as kinematic. This system set new records for continuous autonomous running. Simulating moving the vehicle through different trajectories is at first a reasonable approach, but it is computationally intensive. Simulating all paths is impossible, so one is forced to choose a subset of paths to investigate. One of the advantages which a neural based approach holds over a simulated trajectory approach is that the neural system can be trained by a human expert to quickly recognize image features which indicate appropriate paths. Thus we need not compute the transformation from image to world coordinates for all of the image data; instead we can key off of the salient features in the raw image. Perhaps more importantly, we use color video data to extend the sorts of terrain we can handle.

5.2.4 *Ganesha*

Another CMU system, Ganesha, developed by Hebert, Rosenblatt, and Langer, models the world in terms of regions of constant terrain gradient [Langer93]. The system uses kinematic constraints and uses discrete world obstacles; the system assumes obstacles can be properly modelled. The vegetation issue has not been addressed in this system, either.

5.2.5 *Ranger*

The third cross-country system from the CMU groups is Ranger, which is the successor to the Level I FGN. This system, being developed by Alonzo Kelly has achieved results which include speeds of up to 20 miles per hour in an obstacle avoiding mode. It operates with a perception system and a planner which evaluates a spanning set of possible trajectories based on constraints such as: vehicle roll, vehicle pitch, collisions, terrain continuity, terrain occlusion, maximum path curvature, and continuity of path curvature. This system is optimized for the Navlab II vehicle with the ERIM laser range scanner. [Kelly93] The two possible advantages of our system are that the neural system can be quickly retrained for different vehicle and sensor configurations, and that we handle color video data which enables us to navigate through vegetation. Furthermore, extending our system with new sensors is extremely simple.

5.3 Alignment

5.3.1 ROSA

In the ROSA project developed at CMU several years ago, an inspection sensor had to be inserted into one of many holes in a metal wall of part of a nuclear reactor. The problem had two phases. First a vision system located the hole roughly and then a typical "Peg & Hole" mechanical system finished the job. The vision system implemented was a model-based system using Hough transforms [Ballard82] and a carefully calibrated camera [Coulter90]. On the ANDI project we have a less controlled environment which mandates a more powerful vision approach. We also have a robot which is more flexible and can place the camera less precisely, so we cannot rely on perfect camera calibration.

5.3.2 SM^2

The SM^2 project, also at CMU, involved inserting the end of a flexible manipulator into a hole along the beams of a mock space station lattice [Pomerleau91]. When the end of the leg was close to the hole, control switched to a monolithic neural network which returned the x and y offsets to the center of the hole in the video image. This problem is less complicated than the FAA problem due to a low occurrence of features in the image similar to the desired hole, and in general, the operating environment for ANDI is more cluttered. Furthermore, the features which we are expected to locate are not nearly as regular on a commercial aircraft as on the lattice of the simulated space station. We are responsible for finding different types of rivets with differing degrees of wear and tear, and moreover, the contrast between feature and background is minimal. The SM^2 task is very similar to our TIP inspection mock-up robot in that both systems find a hole in a metal surface. However, each image for SM^2 had at most one target.

5.4 Neural Networks for Navigation

5.4.1 ALVINN

Perhaps the most relevant neural navigation system to our work is one developed at CMU for on-road neural network navigation: ALVINN, developed by Dean Pomerleau. ALVINN typically uses video camera images as input to a monolithic neural network whose output is a steering direction for the vehicle. ALVINN has achieved speed on the order of 100 kph on the highway and has proven itself to be reliable on most any road on which it has been trained. ALVINN has been implemented with the laser range scanner (as its only sensor) and has had some limited success at off-road navigation [Pomerleau92]. Many of our navigation modules are modified ALVINN networks. The results in this thesis will show that monolithic ALVINN networks can be trained to handle multiple sensors very well; however, MAMMOTH frequently offers an advantage of quicker training times.

5.4.2 Carbot

Another system which has some interesting attributes is the Carbot system developed at Indiana University [Meeden93]. Although this system uses a modified toy car, the neural networks used incorporate not just data from a small set of external sensors (two light sensors, and four digital touch sensors for collisions), but it also incorporates internal sensors for measuring the state of the motors. Furthermore, this system was implemented with a recurrent network in which the hidden units from one discretized time unit act as inputs for the next such time unit. The Carbot hints at the potential for sensor fusion and state memory in neural network navigation, but the system and the environment for the Carbot experiments were extremely simple. Our system uses sensors which provide much more data that needs to be processed, and the rapid training times of MAMMOTH are beneficial in such cases.

5.5 Modular Neural Networks

Some problems are naturally solved best when broken into pieces, and the neural network community has been eagerly exploring different approaches to modularity in neural network design. It is worth noting that most of these network designs follow a paradigm in which each module (however it is defined) has the same sort of inputs and outputs. For some of these paradigms, this implies that the modular neural networks are designed for task decomposition as opposed to functional decom-

position. The neural network's answer frequently is chosen from just one of the modules as opposed to being combined from several or all modules. For MAMMOTH, we believe that the right answer can be generated from more than one module and that the modules can have varying inputs and outputs (corresponding to whatever training signal is available in order to learn a particular feature). MAMMOTH is very closely related to many of these systems, but the main differences are that MAMMOTH uses non-homogeneous modules and performs functional decomposition as opposed to task decomposition (as is the case with many of the following architectures). We should note that task decomposition and functional decomposition are not mutually exclusive; we merely state that we are among the first systems to do functional decomposition, and that our way of doing it has benefits in terms of reduced training times.

5.5.1 *Adaptive Mixtures of Local Experts*

Jacobs, Jordan, Nowlan, and Hinton developed a system for multi-speaker vowel recognition based on a modular neural network approach [Jacobs90 and Jacobs91]. Their system is a gated modular system in which each module operates on input of the same type and generates output of the same type. The gating network decides which expert will be applied to the given input case. However, unlike some systems in which the experts are trained by hand, in this system the gating network also automatically allocates different subtasks to different experts for learning. This system is a task decomposition system, unlike MAMMOTH's functional decomposition. This system has an advantage of automatically allocating subtasks to experts; however, in MAMMOTH since we have chosen the sub*functions* ahead of time, we can train them in parallel more easily.

5.5.2 *Pandemonium*

In a similar vein, Smieja and Mühlenbein [Smieja92] developed a system in which the modules compete for different subtasks. The modules, called Minos modules, have two parts. The first part is the worker network which learns to map inputs to outputs, and the second part is a monitor network which is taught to identify input cases for which it has been trained. Thus each module is trained for specific subtasks, and trained to recognize those subtasks. An integrating system, called Pandemonium observes the monitor networks and chooses which expert modules to use based on the confidence reading given by the monitor networks in the minos modules. Again, this system performs task decomposition and has similar advantages and disadvantages compared to MAMMOTH as the Adaptive Mixtures of Local Experts.

5.5.3 *Meta-Pi*

Early work by Waibel and Hampshire [Hampshire89] in phoneme recognition used multiple modules trained to recognize phonemes by different speakers and a speaker identification network (SID) to identify to which speaker the input belonged. The SID network acted to gate the outputs of the expert modules in a winner-takes-all paradigm. The newer Meta-Pi paradigm [Hampshire89 and Hampshire89b] uses the same architecture in the supervisory network differently. Instead of choosing a single expert, the Meta-Pi architecture learns to mix the outputs of the expert networks in a linear combination. Each expert module, of course, has the same inputs and the same outputs. MAMMOTH owes a lot to the Meta-Pi architecture in that the final version of Meta-Pi performs a kind of functional decomposition, allowing novel high level features to be derived from the various modules. MAMMOTH, however, has two major differences: first, MAMMOTH uses a neural network to combine the modules' features, allowing more complex combinations; second, MAMMOTH uses the hidden units of the modules as the basis for combining results (for a discussion of the value of this see "Why the outputs of FeNNs are not best inputs to Task Net" on page 10).

5.5.4 *Connectionist Glue*

In related work, Waibel used a number of homogenous consonant recognition TDNNs (Time Delay Neural Networks), as modules for solving larger consonant recognition tasks [Waibel89]. There are three major similarities to MAMMOTH in this work. First, Waibel's modular networks in this work used the hidden units of the modules to perform the higher level feature combination. Second, Waibel used neural networks to perform the combination. Third, additional inputs and hidden units were added (the connectionist *glue*) to facilitate the combination of features (as with the rivet-finding MAMMOTH network). The major differences are that MAMMOTH uses non-homogeneous modules, and that Waibel's networks in this work still perform a sort of task decomposition. In one example, one of the modules was trained to recognize P, T, or K, and the other was used for B, D, or G, and then the whole net was trained to recognize P, T, K, B, D, or G. In contrast, the MAMMOTH Navlab II network for cross country navigation has one module which learns about mounds of dirt and only has range pixels as inputs and another which learns about grass and only has color video pixels as input, and the Task Net learns to distinguish grass mounds from dirt mounds from flat dirt (which is more of a functional decomposition). Still, MAMMOTH is closely related to this work.

5.5.5 *Cascade Correlation*

An interesting departure from the styles of modular neural networks described above is found in the work of Fahlman on the Cascade Correlation architecture [Fahlman90 and Fahlman91]. In this architecture, there are at first no hidden units. Training commences and when the training peaks, a single hidden unit is added. The weights from the input to the output are frozen. The network's

training begins again with just the single hidden unit, but when learning has peaked, another hidden unit is added. The most significant feature for our purposes is that the newly added hidden unit uses the output of the previous hidden units as input in addition to the inputs generated from the traditional input layer. Taking advantage of previously learned representations is something we use extensively; however, in our case, more a priori knowledge is used to influence the development of those representations. One positive impact of that is that it allows us to learn those representations in parallel.

5.5.6 *Cortico-Hippocampal Model*

Gluck and Myers describe a model which related multiple sensor modalities called the cortico-hippocampal model [Gluck93]. In this method, a multi-layer network maps the input from all modalities to itself through hidden units which act to compress and relate the input stimuli. This is frequently called auto-encoding. In executing a particular task, this sensor representation is affected also by connections to the input which are tuned to this particular task. Connections to the task outputs are fed from the task-specific sensory representations. This method models a possible function of the hippocampus in human learning. Auto-encoding is a not uncommon way to generate compressed representations of input [Pomerleau92]. This approach can be thought of as a MAMMOTH network where the only FeNN is the auto-encoding network, and there are a large number of additional glue units going straight from the different sensor inputs to the Task Net. MAMMOTH, as presented in this work, is a more general expression of this form of network in which the FeNNs can represent many different things (beyond just auto-encoding).

5.5.7 *MTL*

Caruana's recent work in Multitask Learning (MTL) [Caruana96] is in one sense very closely related to MAMMOTH, and in another sense quite an opposite. In MTL for backpropagation trained multi-layer perceptrons, there is one set of inputs feeding into one set of hidden units feeding into *multiple output groups*. Only one of the output groups is the output for the desired task; the others represent associated tasks. The idea is that by allowing related tasks to influence the development of the hidden units, the hidden units become a stronger representation of real world features and not just artifacts of the particular task at hand. Caruana has shown that performance of a given task can be increased significantly with MTL. This is related to MAMMOTH in that the hidden units are useful for multiple tasks (the MAMMOTH FeNNs are trained by signals derived from tasks related to that of the Task Net). MAMMOTH's architecture stands in contrast, though, since the FeNNs can be trained independently. Thus, the main benefit of MAMMOTH is not necessarily performing a given task better, but getting the best performance with less training time.

5.5.8 *MANIAC*

In one of the few fully neural modular approaches to robotics navigation, Jochem has designed a system, called MANIAC, which stands for Multiple ALVINN Networks In Autonomous Control [Jochem93]. Early versions of MANIAC trained individual ALVINN networks on different road following tasks and used a merging network (a la Meta-Pi) to choose the correct steering direction. Later networks used the hidden unit representations from each of the individual road following networks and had the supervisory network learn to drive from those. MANIAC ended up acting as an arbitrator for the individual modules, and each module has the same inputs and the same outputs. In our system the supervising network is not be an arbitrator so much as an integrator of features from different modalities, which is essentially the difference between task decomposition (MANIAC) and functional decomposition (MAMMOTH). An additional difference is that MAMMOTH can use non-homogeneous modules, which makes it well suited for sensor fusion (although MANIAC could be modified to handle this, too). MANIAC also shares with MAMMOTH the ability to train modules in parallel.

5.6 *Summary*

The alignment and navigation problems we address in this thesis are more complicated than many addressed before, especially in the need to fuse data from multiple sensors (as with our Navlab II tasks). The recent crop of modular architectures hold great promise for solving perception problems, and MAMMOTH is closely related to many of them (in particular, Jochem's MANIAC [Jochem93] and Waibel's modular TDNNs with connectionist glue [Waibel89]). Our system's main distinction is that it is designed to be particularly useful for sensor fusion and a wide variety of real-time robotics tasks, and its advantage over most other techniques is the ability to use our a priori knowledge of a task to perform functional decomposition. This can allow modules to be trained in parallel; furthermore, each module may have only a subset of the total number of inputs of the system (a module trained to find range obstacles will not need to have the color video pixels as inputs). All of which means training time is reduced.

6 ANDI

6.1 Rivet Finding

For the FAA's Airworthiness Assurance Research Program, we developed robots to deploy conventional NDI (NonDestructive Inspection) for commercial aircraft inspection. Our prototype robot, ANDI (the Automated NonDestructive Inspector), holds to the aircraft skin with vacuum assisted suction cups, scans an eddy current sensor, and translates across the aircraft skin via linear actuators. CCD video cameras are used to align the robot with a series of rivets we wish to inspect with the NDI inspection sensors.

We addressed the alignment task both on ANDI, which is the actual prototype, and TIP (Test Inspection Platform), which is an in-lab demonstration robot belonging to a different aircraft inspection project with different goals. We developed several methods to solve the task of locating rivets in images of an aircraft skin, including a model-based edge detection system, a monolithic neural network system, and a MAMMOTH neural network system. Originally, we hoped that a MAMMOTH network would prove to solve this problem better than any of the other techniques, but the results do not show any significant performance benefit of the MAMMOTH architecture over the standard monolithic neural network. However, despite some suboptimal training methods,

the MAMMOTH network performed as well as the other techniques which leads into the discussion of MAMMOTH's true advantages in later chapters. With these experiments, we also failed to achieve any advantage in training times due to the network architecture used for the MAMMOTH network, which used a large number of additional inputs to the Task Net.

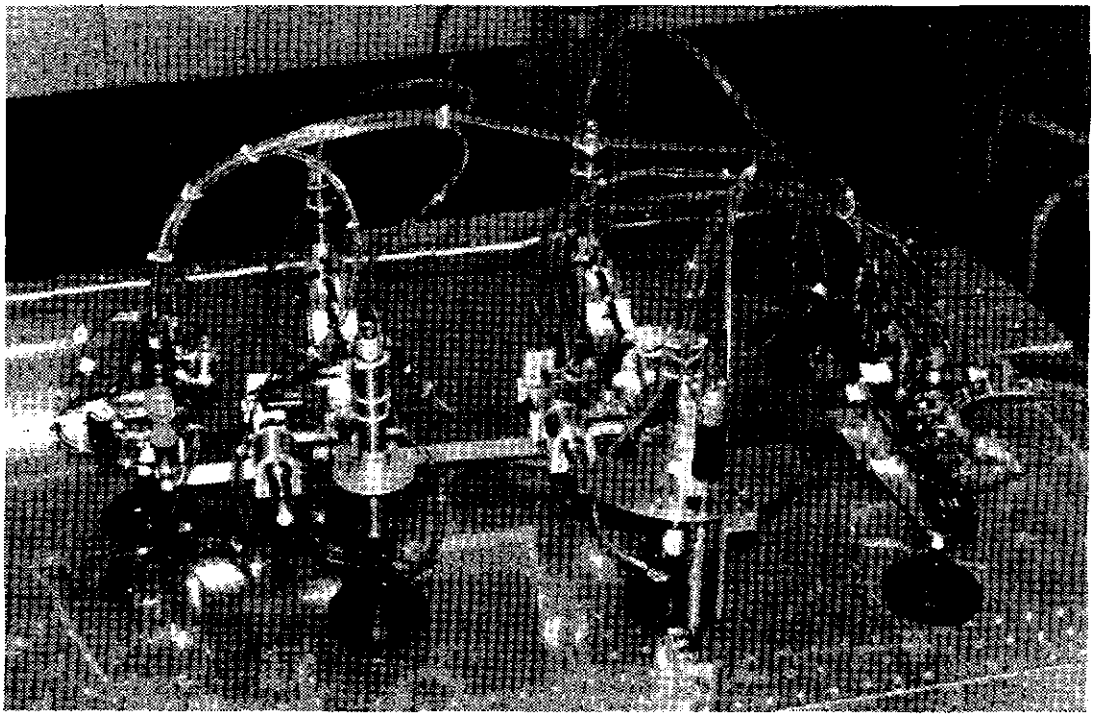


Figure 8 ANDI, the Autonomous NonDestructive Inspector of aging aircraft

An inspection robot must be deployed over a large section of the robot fuselage, and the sensor for inspection must be placed quickly and precisely relative to the centers of thousands of rivets. Towards this end, we must use our video cameras to find the rivets to be inspected. Then we must adjust the robot's position to align the path of its rivet inspection sensor to the rivets.

Our specific task is as follows: we are given a video image, and we must classify the entire image so as to identify the rivets on the aircraft skin. We generate a binary image in which each pixel is classified as rivet or not rivet, and then we use geometric models to filter out misclassifications and to find the best line through the rivets for scanning. It is important to note that the robot is only positioning the sensor; the human operator interprets the data from the eddy current inspection sensor. We also did not take advantage of potentially useful data which could have been gathered from the NDI inspection sensors for our alignment task.

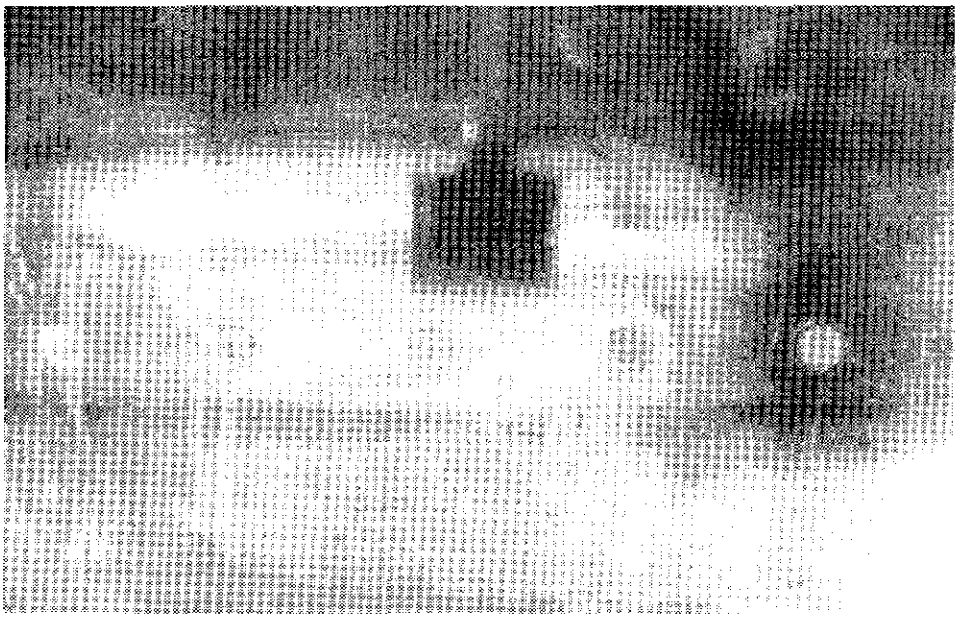


Figure 9 Sample raw input image from alignment cameras

6.1.1 Aircraft Inspection Background

On April 29th, 1988, an Aloha Airlines 737 suffered a catastrophic failure of the integrity of the aluminum skin. In mid-flight, a section of the fuselage above the passengers peeled off, and a flight attendant was killed as she was blown out of the aircraft. The part of the fuselage that peeled off spanned seven rows of the aircraft's passenger seats from the floor level on one side to the floor level on the other side.

The root of this accident was in the pressurization/depressurization cycle that large airliners' cabins undergo. With each cycle, the aircraft skin pulls against the rivets holding it to the structural components. This repeated flexing eventually causes cracking of the aluminum skin. After a large number of cycles, the cracks can join together and form large cracks which can cause the skin to fail catastrophically. An image of the results of this tragedy can be seen in Figure 10 on page 44.

Figure 10 The Aloha Airlines tragedy. A section of the fuselage spanning seven rows of seats and reaching from floor to floor peeled off of the Aloha Airlines 737.

FAA guidelines mandate that the aircraft's skin be inspected with approved methods on a regular basis. The current standard inspection sensor is the eddy current sensor. Eddy current sensors induce a small electrical current in the aluminum to sense the complex impedance. A highly-trained human inspector observes the changes in the complex impedance on an oscilloscope and detects cracks from the shape of the traces as the probe scans over rivets.

The difficulty with this inspection process is that the inspector must use scaffolding to climb onto the aircraft skin for the inspection. She or he must balance the oscilloscope with one hand and slide the eddy current sensor over the rivets with the other. Three primary problems arise. First, the airline must erect the scaffolding around the aircraft which is expensive and time-consuming. Second, the inspector must frequently work in awkward positions. Third, the task can rapidly become tedious and this could potentially result in human error.

By automating the deployment of the skin inspection sensors, the entire inspection process can be sped up significantly. The human operator can perform the complex task of looking for flaws without the tedium of manually deploying the sensor, and if the inspection goes more rapidly, the human error due to boredom can be diminished. Furthermore, by deploying the sensor with a pre-

cisely positioned robot, we can record results from the scans at each location for each inspection and potentially locate problem areas through an analysis of the change in the scan results over time. We may be better able to *predict* cracking.

6.1.2 *The Difficulties of Locating Rivets on Aircraft Skins*

The problem of locating the rivets in a video image of the aircraft skin boils down to detecting metal features on a metal background. In a video camera image, we can see reflections of lights, people, machines, or even a reflection of the video camera itself in the aircraft hangar (see Figure 11 on page 46). The environment is uncontrolled as numerous independent inspections and repairs are occurring at the same time as our NDI inspection. Another major influence on our image quality is that the aircraft that we are inspecting are, by definition, aging. This means that the skin is covered with numerous scratches, scuff marks, and patches on the metal (see Figure 12 on page 47), as well as being interrupted by the doors, windows, and wings that we expect to find on an aircraft. Any surface flaw in the metal, even if not large or deep enough to be worrisome from an inspection viewpoint, will show up on our image.

One important issue in understanding these images is that there are different shapes of rivets with which we must contend. Some are flat-head rivets and some are raised-head rivets (see Figure 13 on page 48), and our image understanding system must be able to detect either sort of rivet for alignment purposes[1]. In addition, rivet placement tolerances are large enough that often even the naked eye can see that they are neither evenly space nor collinear. Thus, we must be able to find rivets in images which contain extreme noise. Many of the subfeatures of rivets are also subfeatures of extraneous image features (such as reflections and scratches). And the rivets themselves are not perfectly predictable in size, shape, or position.

1.Raised-head rivets may be useful for identifying scan lines, but are not inspected with the same type of sensor probe. Thus, in the future it would be useful to be able to classify rivet types.

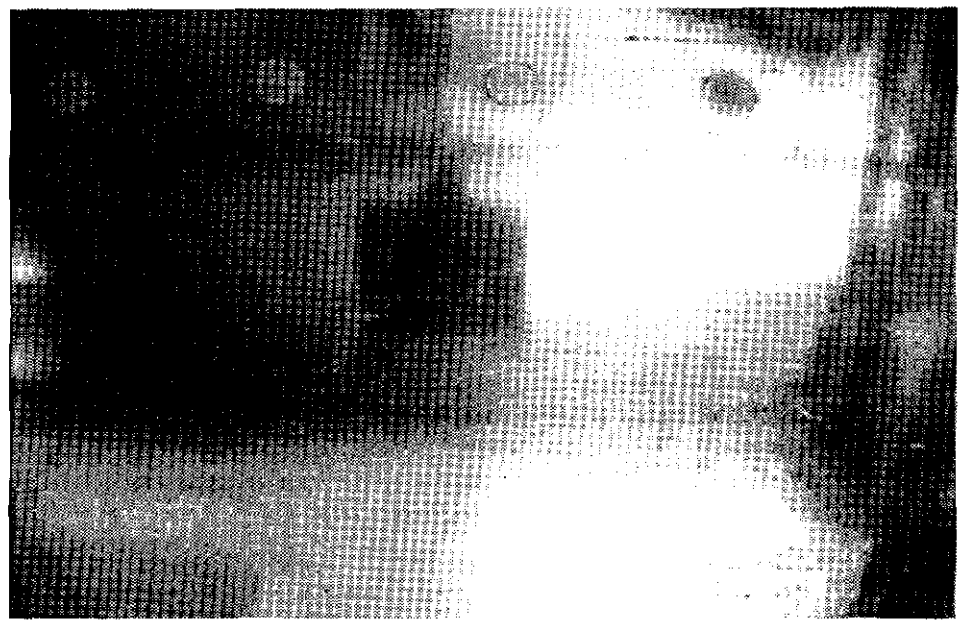


Figure 11 An image taken from ANDI. Notice the reflections of lights, and also of the video camera (just to the left of and above the image center).

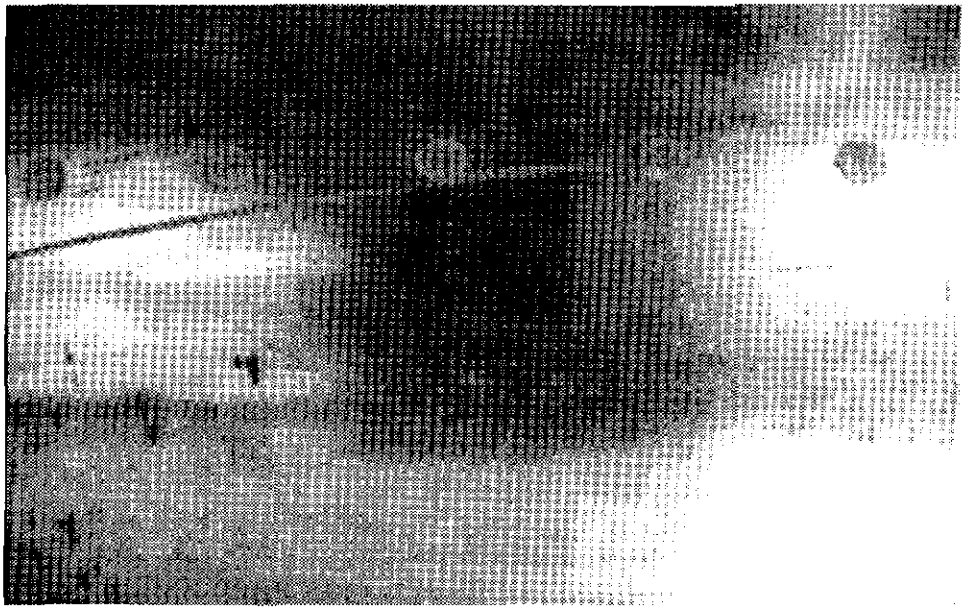


Figure 12 Scuff marks and scratches on aircraft skin.

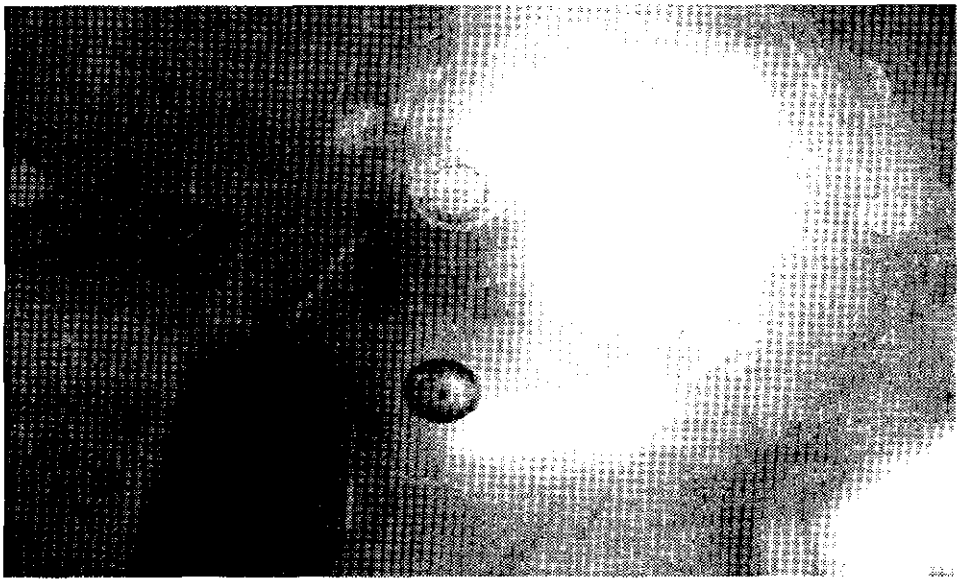


Figure 13 Image taken by ANDI. Notice the raised head rivet below the image center.

6.1.2.1 Artificial lighting limited by power and weight restrictions

To make matters even more complicated, the robot we use (to be described later) has considerable power and weight limitations. These power and weight constraints further limit our ability to control the environment, since we want to avoid mounting lights or shrouds around the video cameras. We have to find the rivets in whatever lighting the hangar environment provides.

6.2 TIP & ANDI

Our rivet finding algorithms have been tested on two separate robots, TIP (the Test Inspection Platform) an in-lab testbed, and ANDI, the aircraft inspection prototype. In this section, we describe each robot. Each of these robots has distinct systems issues, and our experiments with each add different contributions to this work. We used ANDI to demonstrate rivet finding with noisy and difficult images, and we used TIP to show the integration of rivet finding into a complete alignment system.

6.2.1 TIP Inspection Mock-Up

The TIP robot is intended as a development testbed for another aircraft inspection problem being studied in a separate project from ANDI. The TIP system introduces unique implementation issues. The environment, robot hardware, and available computing resources all have differences. The environment was more conducive to the closed loop navigation control experiments than was ANDI's at environment at the time.

6.2.1.1 The Test Site

TIP is an in-lab robot, not intended to ever be used in the field. TIP is a development platform for advanced camera and lighting systems for enhanced visual inspection. As such, TIP has only the rudimentary mobility and controls needed for its task. The test environment for TIP is a tabletop covered by a section of an actual aluminum aircraft skin approximately 2 m x 1.4 m. Realism is high in terms of rivet placement (since this aluminum was on an in-service airliner), but the rivets have all been removed, and the black lab table's top shows through the holes. Furthermore, although the aircraft skin is normally curved, we have flattened this section to allow the TIP robot to navigate with its simple actuation.

6.2.1.2 The Robot

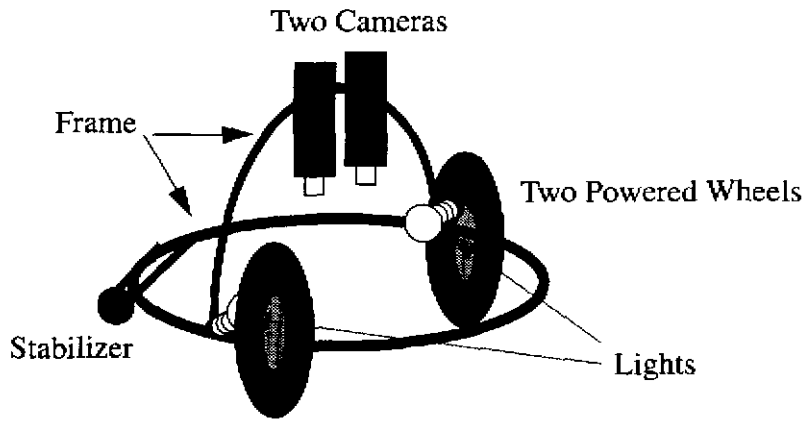


Figure 14 TIP

TIP is a robot with a circular base (approximately 20" diameter), two motorized wheels (and a third stabilization leg), and lights and cameras mounted on board. The cameras are mounted on an arch, and are aligned in this experiment to point straight downwards. Each wheel is controlled by a separate stepper motor.

Motion is made difficult by three factors: tremendous slippage between TIP's wheels and the aluminum aircraft skin, the tendency for TIP's wheels to get caught in the vacant rivet holes, and the frequent entanglement of TIP in its own tether. TIP's controller can accept two types of commands: rotate some number of degrees, and translate some number of millimeters. For all of the difficulties, a typical measured error in rotation is on the order of 2 degrees unless the tether got tangled or the wheels caught on something. Similarly, translations were accurate, in the absence of tether entanglements.

6.2.1.3 The Sensors

Typically, TIP is equipped with two video cameras and several individually switchable lights. There is no global positioning. For our purposes, only one video camera was ever used and it was aligned to point straight down evenly between the two wheels (whose axis was on a diameter of the circular base).

6.2.1.4 Main Loop

The main loop of the TIP system has just three simple steps (see Figure 15 on page 51). Our algorithmic methods for finding the line of rivets fit into the second step.

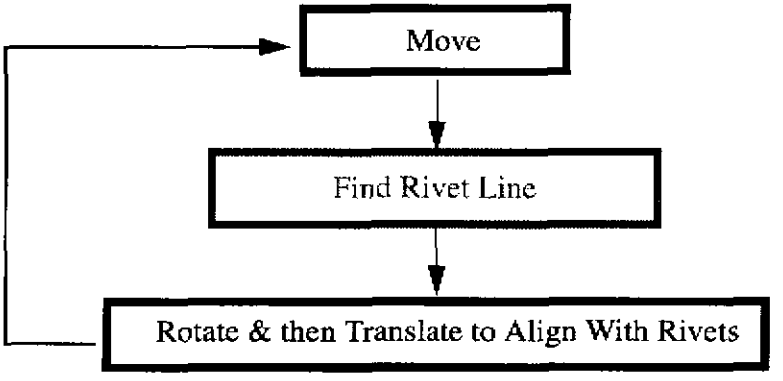


Figure 15 TIP Main Loop

6.2.2 *ANDI Inspection Robot System*

ANDI is the actual inspection prototype robot, which has been used on portions of the carcass of a real airliners for demonstrations. The rivet finding system is only a small part of the entire ANDI system, and this section puts our algorithm in context.

6.2.2.1 The Robot

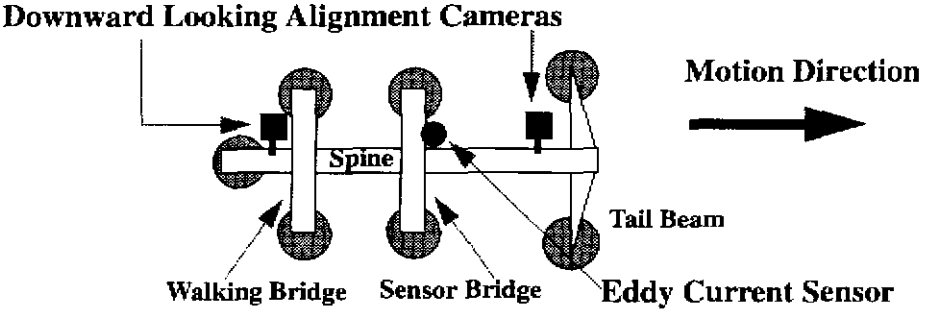


Figure 16 ANDI Diagram. The tail beam is rigidly attached to the spine. The bridges slide along the spine. Full motion description can be found in Figure 17 on page 53.

ANDI affixes itself to the aircraft skin with vacuum assisted suction cups. Electric power, air for the vacuum ejectors, pneumatic actuators, and air-bearings, and input and output signals are transported via an umbilical attached to a safety-tether that hangs from the safety-rail above each aircraft in the maintenance hangar. At the start of an inspection sequence, ANDI is manually positioned close to a manufacturing reference position such that ANDI's main translation axis is aligned approximately along the first line of rivets to be inspected. The primary eddy current sensors (ANDI now deploys an SE Systems SMART-EDDY 3.0 with a Nortec SPO-1958 pitch-catch sliding probe), secondary sensors (e.g., cameras driving monitors on the inspector's console), and system sensors (e.g., alignment, guidance, and navigation aids) are then used to guide the machine alternately in scans over lines of rivets, skin joints, etc., and in walks to and alignments with subsequent sections.

The robot's body is cruciform in design. The main frame consists of a tail beam rigidly attached to a spine in a T-shape. Two "bridges" are mounted crosswise on the spine in the same plane as the tail. Each of these bridges is joined to the spine through a linear actuator that allows the bridge to translate along the length of the spine. Furthermore, each bridge has a lead screw-driven linear actuator perpendicular to the spine, and has a rotational degree of freedom about its point of attachment to the spine. This rotation is normally locked. Suction cups whose vacuum is maintained by pneumatic aspirators are mounted at the free end of the spine, each end of the tail, and at both ends of the two bridges. Walking is standard beam-walking [Mars88]. With the spine affixed to the aircraft, the bridges release their suction cups and are propelled via the linear actuators to the end of the spine. Once the bridges are affixed to the aircraft again, the spine and tail suction cups release and the spine moves through the bridges in the same direction. Once a step is taken, the robot must

be realigned with the rivets for a scan. This process repeats to enable inspection of a large section of an aircraft. A simplified version is shown in Figure 17 on page 53.

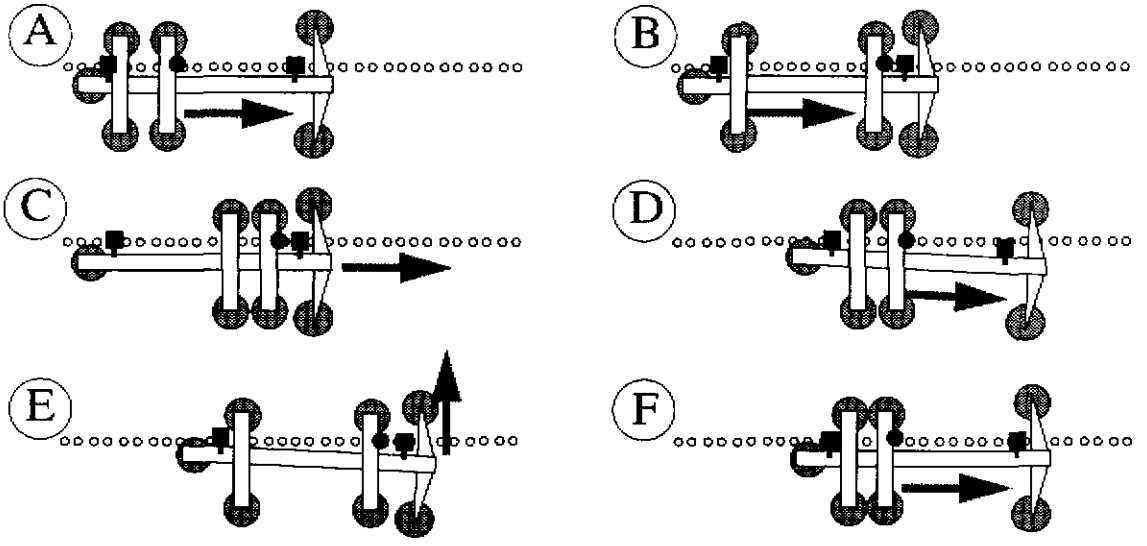


Figure 17 Scanning, Walking, Aligning, & Scanning with ANDI. A) Scan with sensor bridge. B) Step forward with walking bridge. C) Move spine assembly forward. D) & E) Use sensor and walking bridge motors to realign spine. F) Scan next rivets.

6.2.2.2 ANDI Main Loop

The abstract ANDI main loop is more complicated than TIP's main loop since the mechanism is different. However the rivet line finding modules are the same.

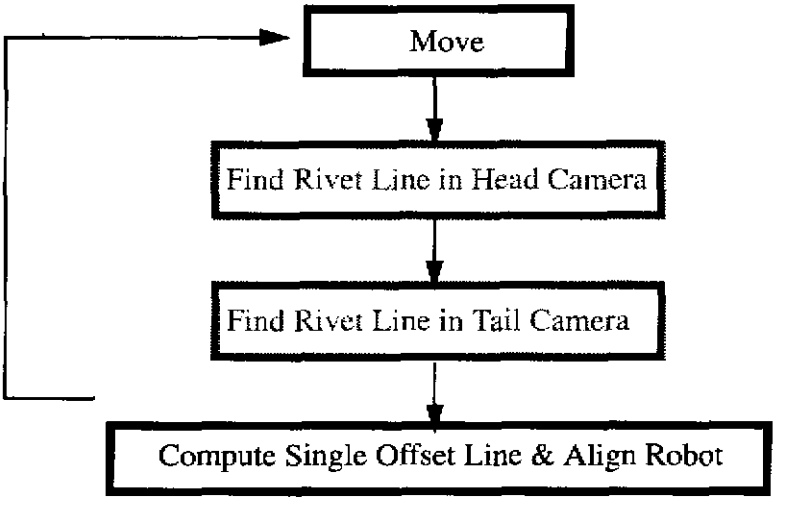


Figure 18 ANDI Main Loop

6.2.2.2.1 Matching Lines in Both Cameras

The main interesting difference between TIP and ANDI (besides the differing methods of locomotion) is that there are two cameras used for alignment, one at the front of the robot and one at the back. These must be well calibrated so that a single line can be computed representing the line passing through the rivets. This computation is not part of this thesis.

6.3 Rivet Finding Methods

Our specific task for ANDI is to find a line passing through the rivets in an image. We know that the edges of rivets are good features for finding rivets in an image. Thus, it was our intention to explore both explicit edge detection as well as neural approaches to solving this problem. Furthermore we felt that using a MAMMOTH network which integrated our a priori knowledge about the usefulness of edges would give us even greater performance. We'll start the discussion of our techniques by answering some of the questions we raised at the end of the chapter "Introduction and Philosophy" about applying a neural network technique to a computer vision problem. Then we shall examine the methods we used in detail, starting with edge detection, then a monolithic neural network, and, finally, a modular neural network.

Should we use an image-local or image-global technique? The features which we are looking for (rivet features such as rivet edges and intensity differences between rivet heads and the aircraft skin) are very small relative to the size of the whole images, so an image-global technique would not have much to work on. Thus, we chose an image-local processing technique. We set the size of the retinas for our techniques to be on the scale of the size of a rivet head in an image. The size of a rivet depends on the optics and the location of the camera relative to the aircraft skin, but in most of the images used in this work, a rivet head is approximately 40 pixels across in a 512 by 480 image. Our local operators will have input retinas corresponding to just larger than the size of a rivet head in order to be able to develop features useful for finding rivets.

What resolution is appropriate? If we were to run a 40x40 operator over every pixel on a 512 x 480 image, processing time would be prohibitive, so we typically reduced the images' resolution to 64 x 60 (a sampled reduction by a factor of eight in each dimension). Then our input to the neural network can be 7 x 7. On a SPARC 20 workstation, the entire image understanding system (see Figure 19 on page 58) can be processed in roughly 3 seconds (with unoptimized code), which is adequate for our purposes.

What is our training signal? The training signal for a rivet finder can be generated by a human classifying a number of different images taken from the robot. The inputs for the training set would be retinas taken from the hand classified images and the outputs would be the binary classification of rivet or not rivet. Care must be taken to balance the training set to be able to find either case.

How do we perform functional decomposition? Edges are very useful features for finding rivets. Therefore, it was our belief that a neural module which locates edges would be an excellent FeNN (although eventually we failed to show that using a MAMMOTH network with this FeNN gave us any measurable advantage over a monolithic network). For this task, we combined the FeNN with additional hidden units attached to raw image inputs (see the section entitled "Modular w/edges" on page 59). The additional inputs served as "connectionist glue" [Waibel89]. The idea was that the

"glue" part of the network would learn extra features besides edges that would be useful for finding rivets.

What are our Virtual Sensors? Our single virtual sensor in this task is the FeNN that finds edge features in a local retina.

6.3.1 Edge Detection

The most obvious low-level feature in finding the rivets on the side of a commercial airliner is the border between the rivets and the sheet metal. Surrounding each rivet is a circular physical break between the rivet and the aircraft skin; although this border is thin, it is visible to the human eye and is detectable in digitized images. The first rivet finding method that we discuss here exploits specular reflections and shadows at these borders. No matter where the light in the image is coming from (with the pathological exception of directly normal to the surface, which could not happen with our camera set-up, since the camera is there), one edge of the border between rivet and skin will reflect the light relatively brightly and the other edge will remain darker. Thus, if we look for sharp changes in the intensity gradients of an image, the edges of the rivets should be apparent. We can look for circular edges of the size of a rivet in order to find the rivets to inspect.

But, scratches are also breaks in the metal skin, and dents and light reflected off the metal frequently look like breaks in the aircraft skin. Even looking for circular edges is tricky because common reflections include reflections of light bulbs which are often circular. Fortunately, we also know how big rivets should appear to us. Thus, we wish to fold our knowledge of the general size and shape of rivets into an edge detection algorithm in order to make a reliable rivet finder.

This method starts with a straightforward implementation of a Canny edge detection operator [Canny83]. For most of our trials, we use a low resolution image version of a high resolution image such as the one shown in Figure 8, "Sample raw input image from alignment cameras," on page 35. Then we run a standard 9 x 9 or 7 x 7 Canny edge detection operator over the 60 by 64 low resolution image separately in each color band. The Canny operator computes the intensity gradients of the images and suppresses those which are not local maxima. The reason for using the operator in each of the three color bands is that actual breaks in the metal, such as rivets, give strong edge readings in all bands. Therefore, we take the results of each of the three edge detection scans and AND them together, so that a pixel is defined to be part of an edge if and only if it is part of an edge in all three bands. This method eliminates a considerable amount of noise.

After the first processing step, the edges are not necessarily connected all the way around the rivets so we grow the edges we have found until they are a continuous region around each rivet. Using just the combined image from all three color bands we grow the regions of edge by using an eight connected grassfire transformation [Duda73]. The primary effect of this is to join regions that are

separated by one or two pixels. This completes the edge circles for many of the rivets in the image. A secondary effect at this resolution is that the circles around the rivets are filled in and the features we have found are solid "blobs". These regions are blobs and not rings because, with the rivet diameters of around 7 pixels, once we grow the edges the centers get filled in (see Figure 22 on page 63).

At this point, we use a recursive connected region extraction method to extract regions from the image, and heuristic methods are used to decide which regions correspond to rivets. This connected region extraction compiles certain basic statistics about each of the blobs. We try to discriminate rivet blobs from other feature blobs using three of these statistics. One is simply the area of the blob in pixels. The second is the aspect ratio of the blob (length/height). The third is the percentage of the bounding rectangle of the blob which is filled (which is highly correlated with the combination of aspect ratio and area, but which can be resolution independent when we later use neural techniques which find the whole rivet and not just the edges). All of these ranges are chosen by trial and error, and have fairly wide acceptable ranges (we prefer to err on the side of accepting a few bad rivets as opposed to missing some good ones). Typical values of these parameters (which vary from data set to data set) are in the ranges of area between 40 and 100 pixels, aspect ratio between 0.55 and 1.7, and fill percentage between 0.60 and 1.0.

Finally, for implementation in a real control system, we find the line in the image which corresponds to the row of rivets to which we want to align the robot. The centroids of all blobs that survive the heuristics are fed into a robust line fitter and a line is fit through their centers (see "Line fitting and geometry" on page 61). The robust line fitter allows us not only to ignore outlying rivets, but also throws out some false rivets. Over the course of our research on the rivet finding tasks, we used two different ways of determining the correct line through the rivets. Each of these is detailed in the section "Line fitting and geometry" on page 61.

6.3.2 *IRA (Instant Rivet Announcer)*

We have to acknowledge, however that the edges may not be the only significant feature of the rivets, nor may it always be reliable. However, with our model-based edge-detection algorithm, we've established a solid framework in which we can easily plug operator modules.

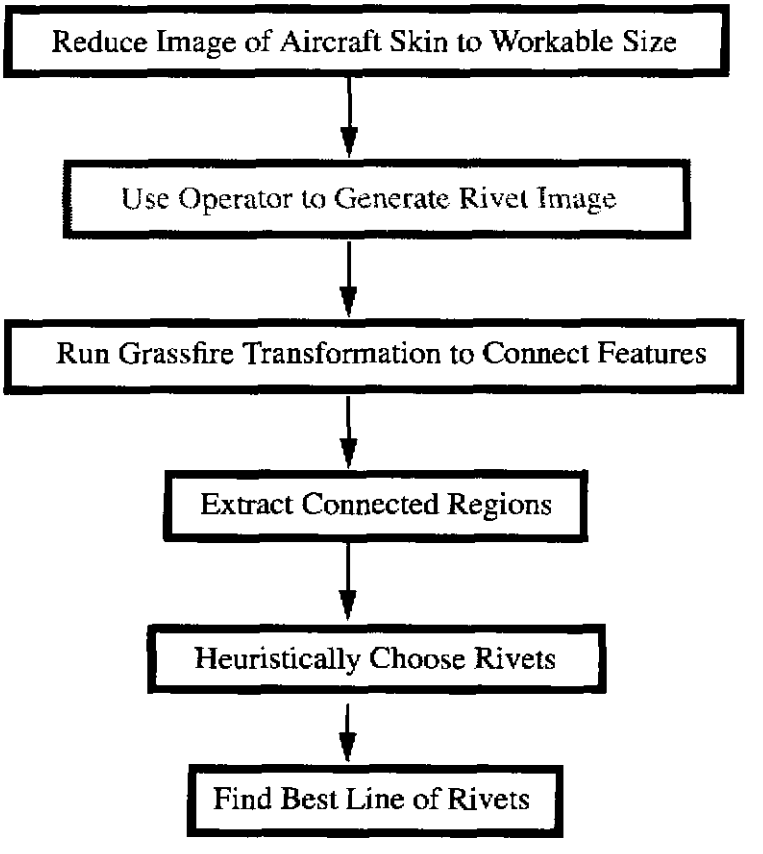


Figure 19 Algorithm Chart of Rivet Alignment Algorithm

Our neural network operators go by the moniker IRA, for Instant Rivet Announcer. For some experiments, we use three color retinas and for some we only use a single grayscale input retina (for speed purposes).

6.3.2.1 Monolithic

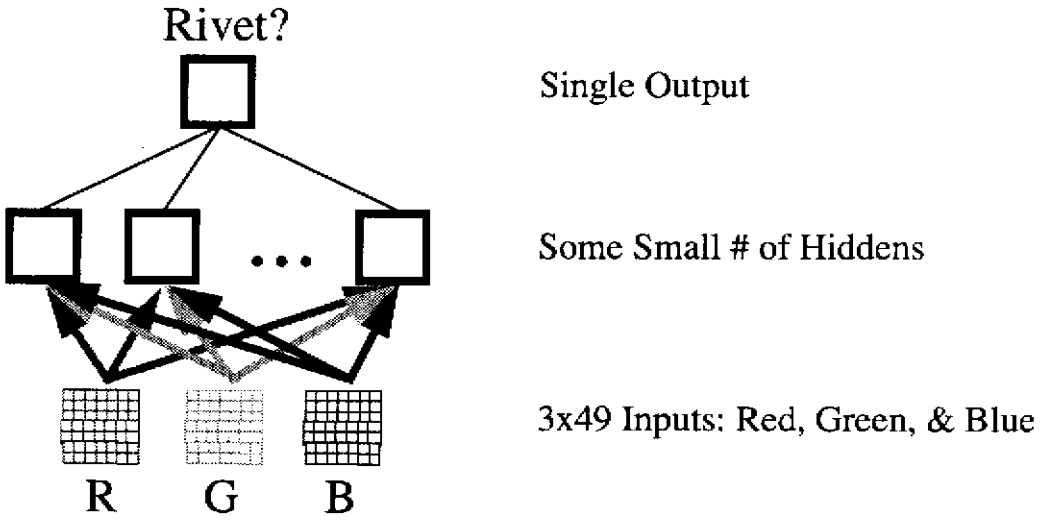


Figure 20 Monolithic IRA Operator Architecture

The important issue in the IRA design is to have the input retina be large enough to encompass a rivet. The output is to read 1.0 whenever the retina is centered over any part of a rivet, and -1.0 when it is not. Clearly, in a noisy environment, this will trigger some false positives, but since the features will be trained to be more specific than an edge detector, the hope is that the output images generated by scanning the operator will be cleaner.

Our training sets usually consisted of close to 1000 exemplars taken from 40 hand classified images. Fifty-five percent of the exemplars showed positive examples of rivets and the rest showed negative examples.

6.3.2.2 Modular w/edges

The monolithic IRA gives us the ability to capitalize on features which we can't predict or model, while the edge detection operator finds specific features that we know are important. Ideally, we would like to combine these two capabilities: the ability to find unanticipated features and the ability to take advantage of predictable features.

Towards this end, we train a MAMMOTH network which has a FeNN for finding edges in images, and which uses a raw image retina as the additional inputs for the Task Net. To speed up processing the input retina we used was grayscale (a color MAMMOTH network would take twice as long to execute as a color monolithic IRA network, but the grayscale MAMMOTH actually had fewer connections that the color IRA).

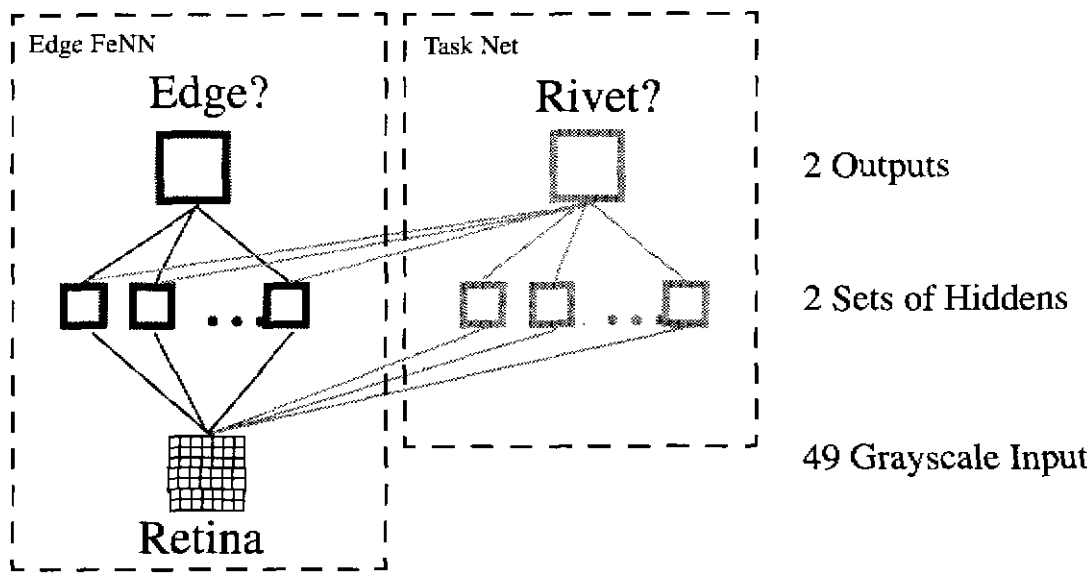


Figure 21 Edge MAMMOTH IRA (Instant Rivet Announcer)

6.3.2.2.1 *The FENN*

The first step is to train a FeNN to find edges in our task domain as well as the Canny operator. Our edge FeNN, shown on the left in Figure 21 on page 60, is essentially a grayscale version of the IRA network (grayscale for speed reasons). It has a retina the same size as the canny operator, and a single output which says whether or not the retina is over a valid edge. For our experiments, the edge FeNN was trained with artificial edge images, about which we had full knowledge of where the real edges were (see "Edge MAMMOTH IRA" on page 68).

6.3.2.2.2 *The Task Net*

Once the FeNN is trained, we freeze the weights internal to the FeNN, and train the Task Net on our Monolithic IRA training sets.

6.3.2.3 Line fitting and geometry

Of course, the final step in the rivet finding is finding a line in the image on which all of the rivets lay. Once our line finder finds the line of rivets in the image, we can calculate the commands needed to align the robot with the corresponding line on the aircraft skin.

6.3.2.3.1 Robust Line Fitting

The original technique we used for finding the line of rivets was a robust iterative line fitter [Press90]. This line fitter minimized the *median* deviation of the points from the line. The idea is that we feed the centers of all found rivet features into this algorithm and it would return the best line for all of the points. By minimizing the median deviation as opposed to the mean, a few outliers have little effect. However, this technique resulted in many bad line fittings in practice because there were few enough rivets in each image that the rivets not on the desired line were not really outliers and could still influence the line placement.

6.3.2.3.2 Line Segment Search

Since the number of rivets in an image is low, and the characteristics of the correct line tightly constrained, we can use a simple directed search to better find the line. We know that the centers of all of the rivets we wish to scan will be very close to collinear. We also know that, most frequently, there will be more collinear rivets on the right line than on any other line in the image.

Our algorithm is a constrained version of a Hough transform [Ballard82]. We imagine a line passing through each pair of rivets and see on which of those lines the most other rivets lie. The differences between this and a naive Hough transform are that we start from a space containing only the centers of the actual image features, and we can limit the number of lines we look at by the number of rivet pairs (n^2) (and then as we check the line formed by a pair of rivets, we can ignore all other pairs on that line). Furthermore, we also have a very good idea of where the correct line is supposed to be since we know where the robot was supposed to go, so we can further limit or direct our search if we so desire.

If there were a large number of features (such as rivets) in the image, the Line Segment Search as presented would be inefficient: $O(n^2)$. Also, if there were a large number of points, the Robust Line Fitting would not suffer from the failure modes that is does. However, at the field of view and resolution that we're examining, at which each rivet can occupy 50 pixels in an image of only 3840 pixels, we do not need to worry much about the computational expense of the Line Segment Search. This is especially true since our heuristics guarantee that the rivets are not connected to each other or overlapping in any way (meaning there is space between them).

6.4 ANDI Experiments

Test Site:	Aluminum simulated aircraft panel (3.0 m by 1.8 m) containing flat-head and raised-head rivets
Training Signal Input:	7x7 retinas taken from 43 images of aircraft panel
Training Signal Output:	Binary hand classifications of images
Training Set Size:	989 Exemplars each for Monolith, Edge FeNN, & Task Net
Metrics for Success:	Count of images in which rivet line was findable for test set of 40 images
Extent of data tested:	40 images tested with best version of each method
Methods Tested:	Model-based edge detector, Monolithic IRA, Edge MAMMOTH IRA (experimentation showed 5 hidden units to be best for the Monolithic IRA, 7 to be best for the edge FeNN, and 5 to be best for the MAMMOTH IRA "glue")
Result of test:	Shows general feasibility of MAMMOTH

6.4.1 ANDI Data Gathering

For this early test, 83 images of a 3.0 m by 1.8 m simulated aircraft panel (which is primarily used for tests of the robot mechanism) were digitized. The images used have both flat-head and raised-head rivets, have outlying rivets (not on the main scan line for the eddy current sensor), are not necessarily centered over the line of rivets, and have uncontrolled lighting. Of the 83 images, 40 were used to test our algorithms and 43 were used to generate neural network training data.

Each image was converted from a high resolution (512 x 480) color image into a low resolution (64 x 60) image for real-time processing (the high resolution images contains 64 times as many pixels to process). For the MAMMOTH technique, we used a grayscale image, and for the other techniques we used a color image. We tested the model-based canny edge detector method, the monolithic IRA network, and the Edge MAMMOTH IRA on the 40 test images. All of the methods for locating rivets that we discuss here operated on the order of one image every 4 to 6 seconds (including digitization and conversion to low resolution) without optimization.

The robust line fitter we used to fit rivet centers to lines at this early date was prone to failures, so the error metric we use here is not lines successfully fit, but images in which enough rivets were found to fit a line through the centers of the actual rivets (this was recorded by a human observer; these "positive" images were either those in which the line fitter found the correct line, or those in which the only rivets found were on the correct line, but the line fitter failed). The results for each of our techniques on this small experiment are presented below.

A point important to mention is that in this early neural network research (and just for this set of experiments), we did not integrate into our neural network training code a way to measure error on

an independent test set when we trained the networks on their training sets (no cross validation). This can make the performance of the networks suboptimal on new data, as networks can overlearn their training sets (meaning they learn a classification or mapping based on noise or extraneous features of the training set). With the small number of hidden units and weights to be trained in our networks, the risk is somewhat diminished, as small nets force generalization to some degree. The positive side of this is that our neural network techniques still performed very well despite this sub-optimal training method (perhaps because the low number of weights and hidden units forced some generalization). We did train nets with different numbers of hidden units, and report the results from the nets which performed best on our test images (which were distinct from the training images).[1]

6.4.1.1 Canny Edge Detector

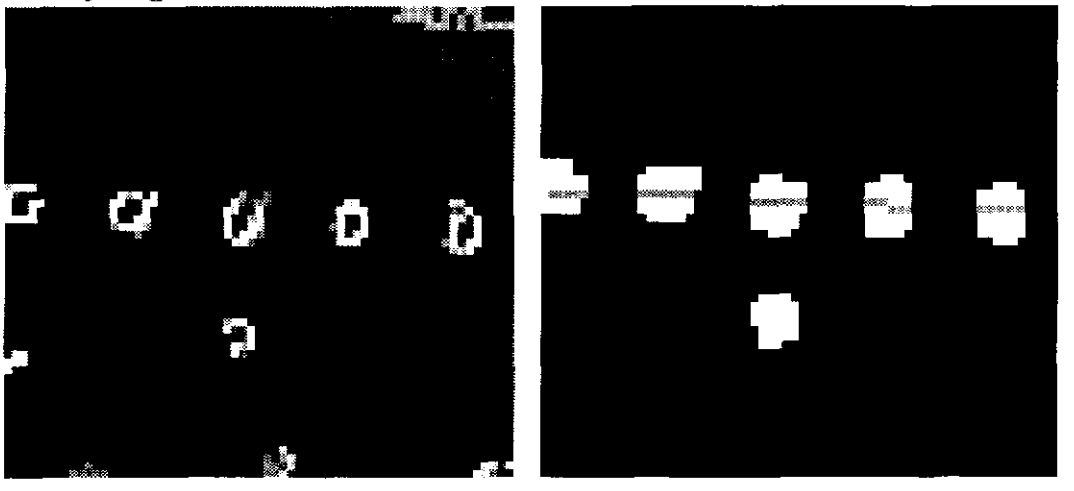
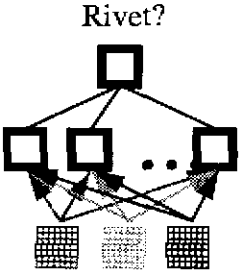


Figure 22 Edges found with Canny operator & rivets and rivet line

Our first method tested was the model-based edge detector. Although the output images generated by the canny edge detection method are very good when they're good, the method is less robust to lighting and scratching issues than the later neural techniques which key on more than just edge features. On the 40 test images, we failed to find enough rivets on *six* images.

1.An equipment failure (storage medium) caused many of the images from these tests to be lost before our final neural network system was implemented.

6.4.1.2 Monolithic Retinal IRA



Our training set for the Retinal-IRA neural network approach was generated from the 43 images reserved for training the neural networks. We randomly chose 989 retinas (23 from each of the 43 images) and their corresponding classifications (based on whether the center pixel of the retina had been classified as part of a rivet or not) as our training set.

Training progressed for 10,000 epochs until the error rate on the training set stopped improving (see the note above on why this is overtrained). This relatively long time is due in part to the complexity of the task (poorly illuminated images made for a low dynamic range in the images), but even on a relatively low-end workstation by today's standards (a SPARC II IPX), the entire computing time was less than thirty minutes. With 10,000 epochs and 989 exemplars in the training set, the total number of exemplars seen was 9,890,000.

In the test images, we scanned the IRA neural operator over each pixel (except for those within 3 pixels of the edge of the image). This required 3132 forward passes of activation propagation through the network per image. The resulting intensity image was 58x54 as we did not center the retina over pixels which would require part of the retina to be off of the image. The entire test set of 40 images represents 125,280 applications of the operator.

In the 40 test images, the neural-based image understanding algorithm failed to find enough rivets to form a good line in four images.

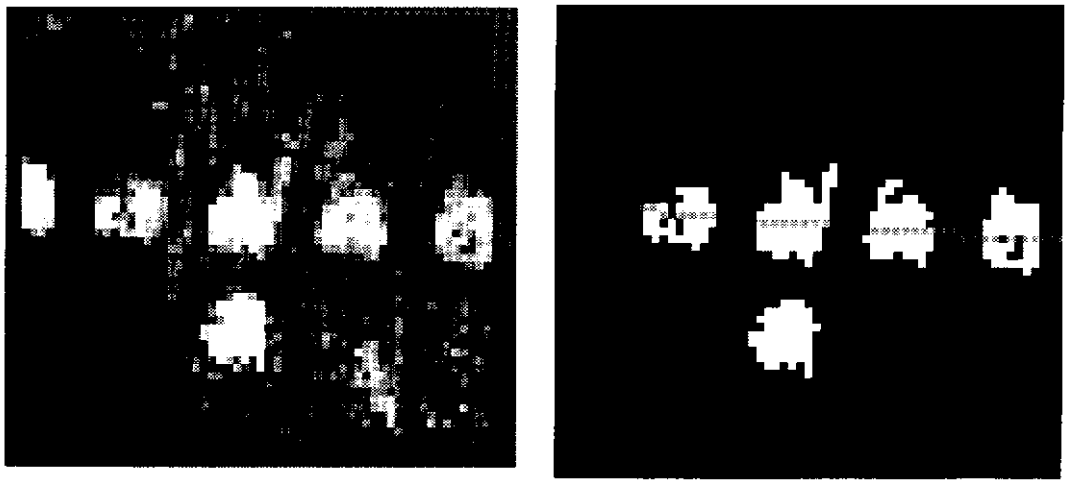


Figure 23 Intensity image generated by neural rivet finder & rivets and rivet line. Notice that the left-most rivet is cut off by fact that we did not run the neural rivet finding operator over pixels very near the edge of the image. Our heuristics for filtering out non-rivets includes a range of acceptable aspect ratios for rivets, and the "half rivet" at the left edge just fails that test.

6.4.1.3 Neural Edge FeNN

Construction of the training set is the main issue in the design of the neural edge FeNN. For purposes of comparison we desired the network to find similar edges as the Canny operator, so we trained the FeNN to find general edges.

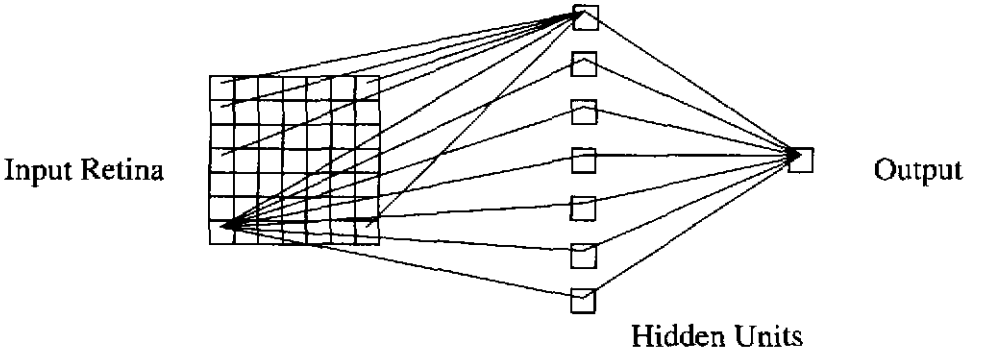


Figure 24 Edge detection FeNN

Our training set for the neural edge detector was made to be a series of artificial edge images. We constructed a training set of nine hundred 7 x 7 images, of which roughly 25% contain no edge at all, and had just a blank background of a random intensity. All of the remaining images had an edge that was positioned randomly. In half of these the edge was a line drawn over a background of a random intensity, and in the other half the edge was the meeting of two patches of different intensities at a line. In two thirds of the "edge" images the edge was close enough to the center of the image for the output of the operator to be targeted as "edge" and in the other third the edge was far from the center and the output was targeted as "no edge." Therefore, approximately half of the images were positive instances and half were negative instances. Noise was added to each training image in an attempt to make the operator more robust in the presence of noise in the input images (each pixel's intensity could vary either positively of negatively up to 4% from the background intensity). See Figure 26 on page 67 for several examples of artificial edge images.

Training progressed for 2000 epochs, at which point the learning algorithm ceased to improve the network's performance on the training set (again, this network may have been overtrained). To provide some sort of qualitative comparison between the neural edge detector and the Canny edge detector, we simply plugged the neural edge detector into the algorithm for finding rivets in place of the canny edge detector. In our usual 40 test images, we failed to find appropriate rivets in only two images. There were three fewer failures on the test images than with the Canny edge detector, and

although we can hardly generalize from such a small experiment that the neural edge detector is in any quantitative way superior to the Canny operator, it is shown to be sufficient for our FcNN.

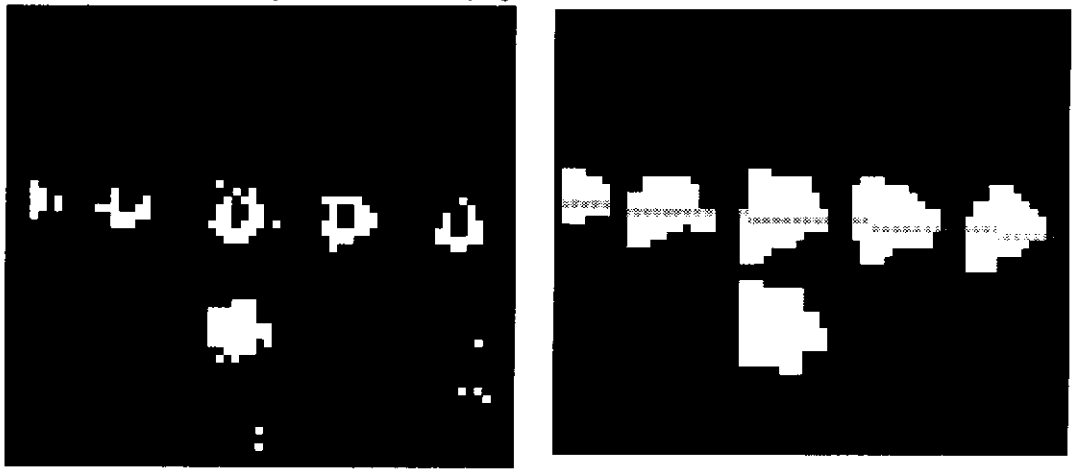
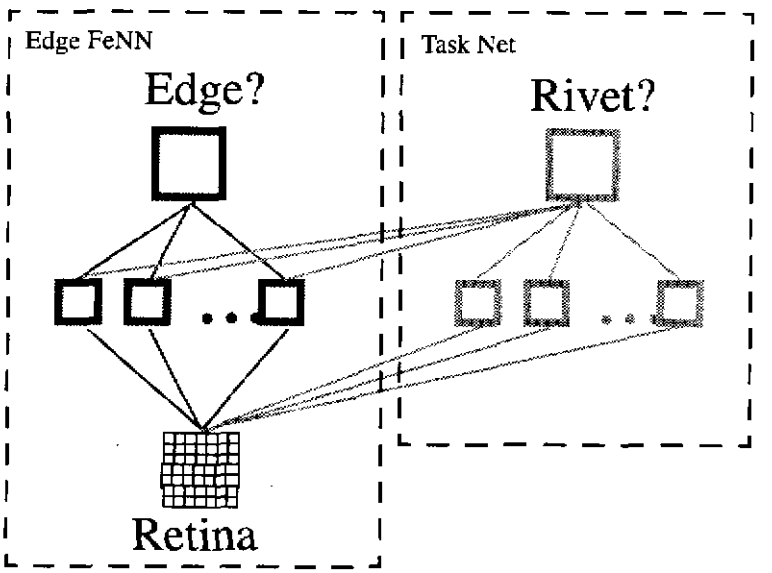


Figure 25 Intensity image generated from edge FeNN & rivets and rivet line. Although the edge image is not as clean as that for the Canny, note that the previous image showing the output of the Canny operator shows the output combined for all three color bands, while this shows edges found only for grayscale, so a direct comparison is not fruitful.

Figure 26 Sample artificial edge images

6.4.1.4 Edge MAMMOTH IRA



To train the MAMMOTH-IRA, the edge FeNN was trained as above (see "Neural Edge FeNN" on page 66), the internal FeNN connections were frozen, and the Task Net was trained. The Task Net was trained with the same training set that was used for the monolithic IRA network. The complete MAMMOTH-IRA network was tested on the typical 40 images. In the 40 images, the Edge MAMMOTH IRA system did not once fail to find enough rivets for a good line fit. Unfortunately, with so few images, we cannot claim any general superiority of MAMMOTH over the monolithic neural operator, nor can we see any major qualitative advantage in the Edge MAMMOTH intensity images. The only notable result we can derive from this is that the MAMMOTH technique did not do *worse* than the monolithic technique on our small set of images. The feasibility of the MAMMOTH approach is all we demonstrated on our 40 images and 125,280 applications of the operator.

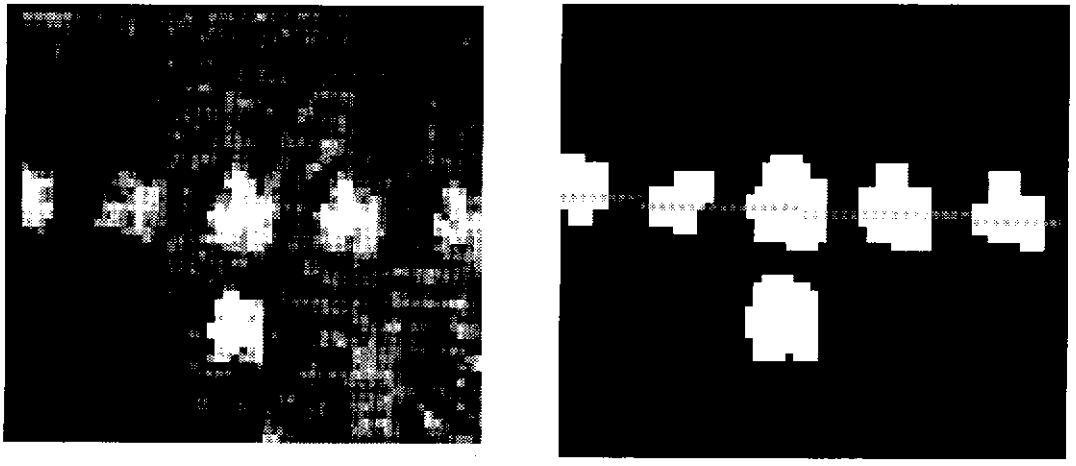


Figure 27 Rivet intensity image generated by MAMMOTH-IRA & rivets and rivet line

6.5 TIP Experiments

Test Site:	Aluminum panel from actual airliner (2.0m x 1.3m) containing only rivet *holes*
Training Signal Input:	7x7 color retinas taken from 40 images of aircraft panel
Training Signal Output:	Binary hand classifications of images
Training Set Size:	1000 Exemplars each for Monolithic Color Retinal IRA
Metrics for Success:	Ability to find rivets Ability to find lines Ability to navigate properly to waypoints
Extent of data tested:	37 images per test run. 4 test runs
Methods Tested:	Monolithic Color Retinal IRA (6 hidden units, as reported by ANGEL)
Result of test:	Shows integration of ANDI-style vision system into complete navigation system

Although the brief rivet finding experiments on ANDI did not show any particular advantage of MAMMOTH, our exploration of the aircraft inspection problem led us to put together a test system on which we could demonstrate the integration of our vision system into a complete robot control loop. Thus, the next step in the rivet finding work was to experiment with a robot performing the actual alignment. We used the TIP robot as it was a more convenient platform on which to test our algorithm in a complete control loop (using an improved line fitting algorithm).

The goal of our TIP tests was to move to six different waypoints and align with each of them. To make the task more challenging, after moving to each point, we rotated the robot up to ninety degrees so that the line of rivets would not necessarily be near the center of the image. The path taken by the robot is shown in Figure 28 on page 70.

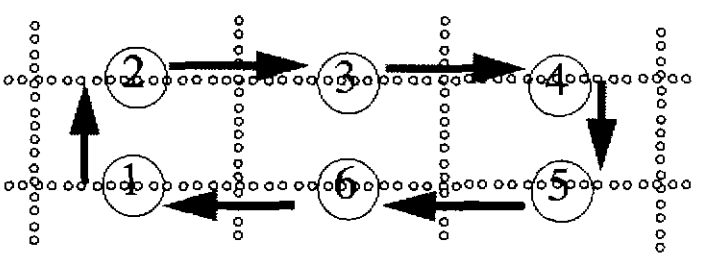


Figure 28 TIP's path. The robot moved to each of the six waypoints, performed a random perturbation *at* each waypoint, and then aligned with the rivets at each waypoint.

6.5.1 *The Data, the Network, & Training*

Although TIP's vision environment was an actual aircraft skin (as opposed to a simulated aircraft skin as in the ANDI experiments), the rivet holes were rivetless, which made the general rivet finding problem significantly easier. The skin still had scratches, specular effects, and skin joints. For TIP, we used the same Color Retinal IRA which was also used for ANDI (see "Color Retinal IRA" on page 71). Forty hand classified images were used for training set construction. Twenty-five retinas were used from each of the images for the training set and for the test set (on this experiment we used the ANGEL training system which found the best trained network based on an independent test set; see "General Neural Techniques" on page 12). The neural network converged to its best results on the test set in fewer than 1000 epochs, and ANGEL reported the best network to have 6 hidden units.

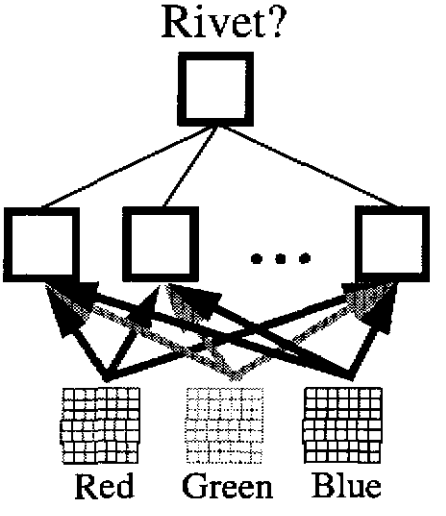


Figure 29 Color Retinal IRA

6.5.2 *Rivet Hole Finding Results*

In each successful full length run, we navigated through the six waypoints and aligned to them. We repeated the experiment six times total. This required 37 alignment images (we aligned with the waypoint 1 an extra time, since we aligned at the beginning and at the end). In the 37 alignment images, each of which contained 5-7 rivet holes, we found every complete rivet hole (some partial

rivet holes were cut-off at the edge of the image). We found a handful of false rivet holes, too, but fewer than 1 per image.

6.5.2.1 Line Finding Results

Because of the disappointing results of the line fitter from the early ANDI tests, we used our line-search algorithm for this task (see "Line Segment Search" on page 61). In the 37 images in each successful run, we found the correct line of rivets *every time*. The difference between the degree of rotation suggested by our line-finding and the commanded random rotation was never more than 2%, and this was caused largely by slippage of the wheels in performing the random rotation (the commanded rotation was fairly consistently larger than the observed rotation).

6.5.2.2 Navigation Results

At the end of the 37 waypoint trip, we ended up with the robot within an inch of its starting point, and perfectly aligned with the rivets at the start point. In the course of a week of testing we performed this complete test four times. The only failures we encountered were when the robot would slip or hit its tether, causing it to fail to move properly to the next station. This caused three additional testings to fail to proceed through all 37 waypoints. However, there were no failures of the vision system.

6.5.3 MAMMOTH & TIP?

For TIP, there was no need to try a MAMMOTH network, since the monolithic network worked as well as could be hoped for and the Edge MAMMOTH IRA had the additional overhead of the time to train the FeNN. Thus, we could expect neither better rivet finding nor a reduced training time.

6.5.4 Additional ANDI results and data

We have also collected two additional data sets from the ANDI test panel. The first was gathered with very controlled lighting (with skirts around the camera to shield the field of view), and in the tested images the rivets are as easy to find as with TIP. The second has very bad lighting and was gathered with bad cameras and a poor digitizer, and to date none of the techniques has yielded acceptable results on this data set.

6.6 Results

The following table recaps the results for the ANDI experiments.

Technique	Images in which line could be fit
Canny Model-based Edge Detector	34/40
Monolithic IRA	36/40
Edge FeNN	38/40
MAMMOTH IRA	40/40

And the next table recaps the TIP results.

Metric	Success rate
Successful Alignments	148/148 (four tests of 37)

7 Navlab II

7.1 Navlab II, or HMMWV (High Mobility Multi-Wheeled Vehicle)

Although the results of MAMMOTH for ANDI were inconclusive as to any performance or training benefit of MAMMOTH, the feasibility of MAMMOTH was demonstrated. With the Navlab II tasks, we found a true benefit of MAMMOTH: decreased training time with equally good performance.

Our final tasks all involve low-speed multi-sensor reactive steering for a four wheeled vehicle. Reactive steering means we map directly from the sensor data to a steering command for the vehicle without any memory. Reactive navigation is desirable at times because a reactive navigator can operate without a map or any other "Top-down" data. But for reactive navigation to work, having rich sensing is crucial. *It is important to note that our systems control only the steering direction of the vehicle; speed is controlled by a human safety driver (the driver tried to hold the speed relatively constant at speeds between 1 and 3 meters per second).*

Figure 30 The Navlab II, or HMMWV

Many current experimental systems for both road navigating (RN) and cross country (XC) navigation tasks utilize only a single sensing modality to accomplish some task. Road following has been done with just a single CCD video camera [Pomerleau91] and obstacle avoidance has been done with a single laser range finder [Kelly95]. Here, we examine tasks *requiring* multiple sensors.

7.1.1 *Multi-sensor Modalities in RF & XC*

For real-time navigation with a four-wheeled robot, we need sensors that will very quickly provide enough information about the world to navigate. In road following tasks, the sensor of choice has been the CCD video camera. Using video intensity data, a road follower can find the edges of the road, the lines painted down the center of the road, and other features pertinent to staying in a lane, such as oil stains down the center of a lane [Pomerleau95]. However, when there are objects in the road such as garbage, animals, or other vehicles (parked or moving), a single video camera is not the best sensor for finding these essentially range-based obstacles.

In contrast, for cross country navigation tasks, the sensor of choice has been some type of range imager. Stereo video cameras and laser range finders are used to generate images in which the intensity at each pixel corresponds to the distance from the sensors to the object at that pixel in the

image. But range sensors are notoriously bad at being able to determine what sort of object generated that range signal; for example, a range sensor cannot be used to tell the difference between a rock and a bush with the same physical dimensions. A truly capable cross country navigation system must know when an object is made of vegetation which can be driven over as opposed to rocks which cannot. See Figure 31 on page 77 for examples of real and simulated range and color video images.

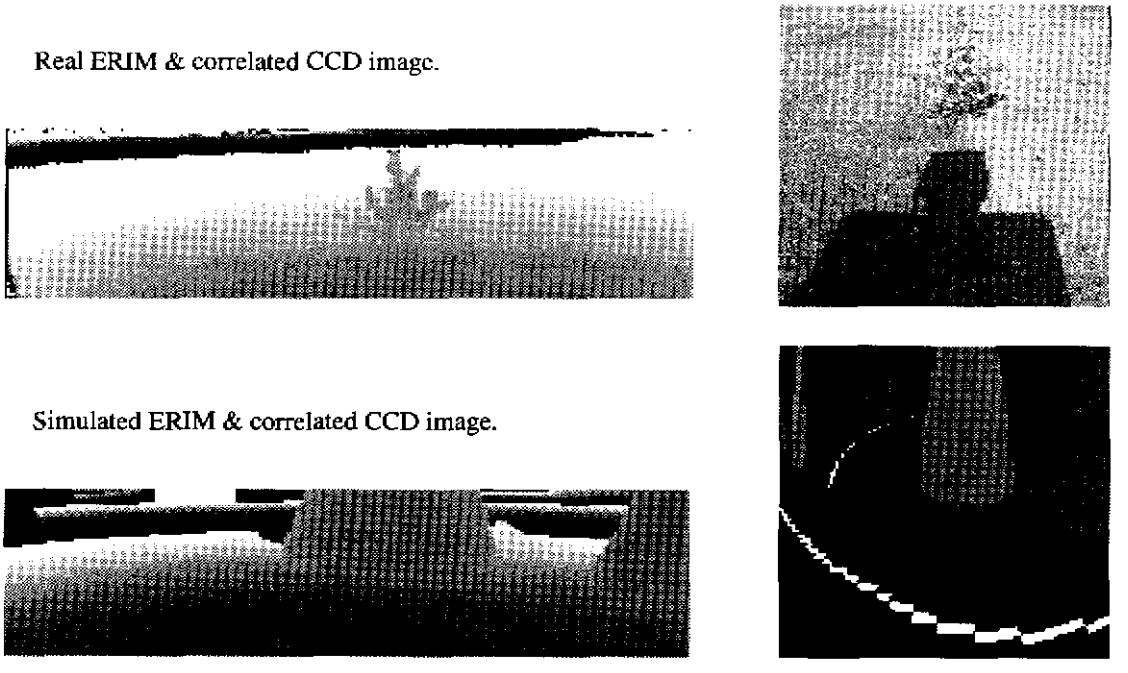


Figure 31 Real & simulator ERIM laser range and color CCD images

7.1.2 *From Road Follower to Road Navigator*

A strictly reactive road follower will have a hard time avoiding arbitrary obstacles of the sort that might be encountered on a real road if it is restricted to monocular vision. Range images can provide the extra information needed to turn road following into road navigating. Our goal is to turn a road follower into a road navigator by using both CCD video images and range images for navigation. The inputs to our system are range and video images, and the outputs are steering commands.

One of the nice things about road following has always been that (except at intersections), we always know where we should go. For a system that will follow a road, having a relatively low number of obstacles in the road still leaves us with a fairly clear notion of where to steer - all we need is the added sensing information for locating the obstacles. Our task is to take a range image and a color CCD image and steer so as to stay on the road, but also so as to avoid the obstacles which would be dangerous to the vehicle.

7.1.2.1 Static obstacles

Our initial goal is to perform road navigation in the presence of static obstacles. Static obstacles might be parked cars, dead animals in the road, or other natural obstacles such as fallen trees or rocks. The obstacles will be large enough to require significant path adjustment, but sparse enough that we still have one obviously correct steering direction. For a strictly reactive system, there is some similarity between arbitrary static obstacles and moving obstacles - although the system will behave overcautiously for obstacles it comes up on (assuming they are moving in a similar direction to the vehicle). Only if we have velocity information for obstacles can we be a bit smarter, and as that is beyond our reliable sensing capabilities right now we'll only examine static obstacles.

7.1.3 From Obstacle Avoider to Cross Country Navigator

In cross country navigation, we have a slightly different problem from road navigation, but this problem has a very similar solution. An XC system which can handle vegetation intelligently will be able to reach a larger portion of a real-world test site. Hence, adding in color video images to an XC system gives us enough information to allow us to navigate in realistic scenarios. As with our road navigator, the inputs to our system are range and video images, and the outputs are steering commands.

7.1.4 The XC Tasks

In cross country navigation, the correct steering direction is a bit less obvious than in RF applications, as there may be several acceptable steering directions to avoid a given set of obstacles. But by deciding on a steering "algorithm", we can solve many XC tasks with similar techniques to those used in RN. For this work, we'll examine four navigation tasks that build up to a system which solves a new level of XC navigation problems.

7.1.4.1 Simple obstacle avoidance

The first thing we will show is that we can avoid obstacles using techniques similar to the road following techniques (ALVINN style networks [Pomerleau91]).The goal is a "wander" mode system which controls just the steering direction of the vehicle. The system will not seek particular goals, but will avoid large obstacles that would be dangerous to the vehicle. Our desired steering algorithm for avoiding a single obstacle is simply to turn left if the obstacle's center is in the right half of the image and vice versa. At our test sites (real and simulated), there is typically be enough room between any two discrete obstacles for the vehicle to fit, which makes handling multiple obstacles simpler (you can sum the steering directions as a reasonable first order algorithm). That is only our desired steering algorithm, and getting the system to do this is the interesting part. In fact, our neural network techniques have shown an ability to avoid obstacles that are too close together (to be discussed later).

7.1.4.2 Simple vegetation awareness (avoidance)

Our second task is designed to show that we can use the color video camera to find vegetation features related to navigation. As above, we'll show that we can train a network to recognize and avoid obstacles in a single sensing modality: in this case, color video. Although this vegetation avoidance is, in itself, novel, the structure of the problem (large obstacles with space between them) and the techniques used will be the same as for the simple obstacle avoidance.

7.1.4.3 Additive obstacle task

In order to demonstrate new cross country navigation capabilities as well as a new sensor fusion technique, we next show a system that needs both main XC sense modalities (laser range data and color camera data) in order to accomplish its goal. This system is a reactive navigator which has a simple goal: don't hit big solid obstacles and also don't hit any vegetation. We call this task the additive obstacle task because second sensor only adds new things to avoid. The sorts of obstacles that will be present in both sense modalities include grass patches, mounds, ridges, cliff walls, trees, and other natural vegetation.

7.1.4.4 Conflicting obstacle task

Our final task is one in which the data from the two sensing modalities must be fused in a more complicated manner. This task is called the conflicting obstacle task because the data from one sensor "conflicts" with the data from the other. That is, the vehicle will be driven through an obstacle course in which the main obstacles to avoid are mounds of earth, but there are also patches of vegetation that are roughly the same dimensions as the mounds of earth. In the range sensor modality, both the dirt mounds and the vegetation look the same. However, we can use the data from the color video camera to "override" the range interpretation. For this task, the obstacles are close

enough to each other that we have to run over the vegetation in order to pass through the obstacle course at all (for this task, the robot is not trained to avoid vegetation ever; the next logical system would try to avoid vegetation when there is an opportunity).

7.2 *Navlab II SYSTEM*

The Navlab II tasks are performed either in the real world or in our simulator, and while the intent of the simulator is to allow us to develop techniques that will work in the real world, each environment has it own issues that must be dealt with.

7.2.1 *Malsim {Modified Alsim}*

Our simulator, Malsim (Modified Alsim [Kelly95]) allows us to test our methods extensively before taking out the real vehicle. The simulator gives us both freedom from hardware problems (the generators, sensors, or the vehicle itself are prone to failures) and the ability to control the environment in which we are performing tests.

Malsim is a 2.5D simulator. The Malsim world consists primarily of an elevation map of the world, a color map of the world, and a visual elevation map of the world. The elevation map is used for determining the z position (height) of the vehicle for each of the four wheels. The color map and visual elevation map are used for the simulated sensors. The simulator can simulate a vehicle moving through this world and generate simulated sensor images (color video or laser range) through simple a ray tracing algorithm. Having the separate elevation map and visual elevation map allows us to have "obstacles" in the sensor images which can be seen in the simulated images, but which can be driven through, as would be the case in the real world with bushes or high grass.

The user can write a world description file which can include any number of features such as mounds, blocks, roads, pits, and more. Each world feature can be of varying size and has a corresponding algorithm for generating the elevation, color, and visual elevation of each cell that the feature occupies in the world. See Figure 32 on page 81.

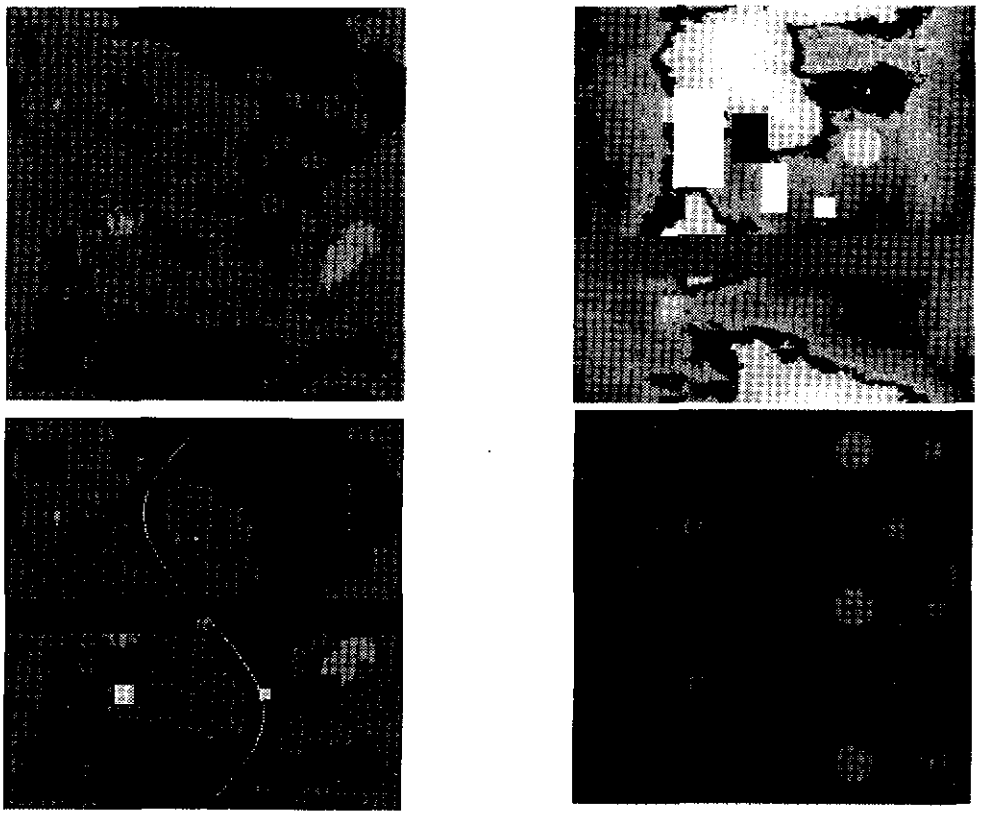


Figure 32 Several Sample Malsim Worlds (Two color maps on the left, two elevation maps on the right).

Control of the vehicle is through a routine with the same calling parameters as on the real vehicle. The control routine provides a steering radius and a speed. The main Malsim Loop is shown below.

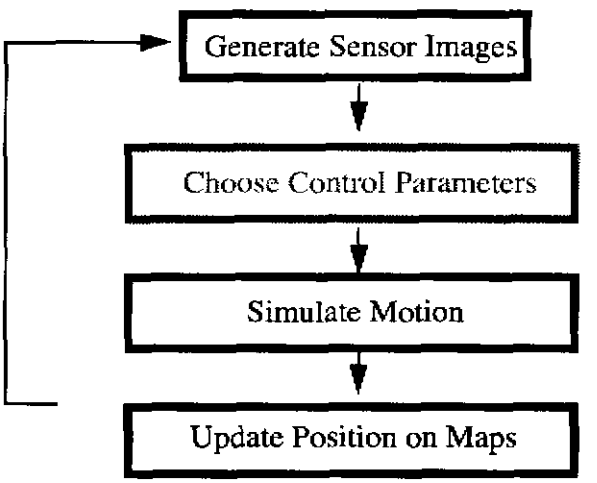


Figure 33 Malsim Main Loop

7.2.1.1 Omniscient Driving Algorithm (ODA) & Improved Omniscient Driving Algorithm (IODA)

One of our modifications to Alsim is the addition of an omniscient driver for training XC neural networks. With the ODA, the user can specify a number of points that act as repulsors for the vehicle. When new images have been digitized, the ODA computes which of the repulsors is within the field of view of the sensors. Each repulsor influences the vehicle to turn away from it proportionally to a linear combination of the inverse of the lateral distance and the euclidean distance between the vehicle's front center point and the repulsor's center (the exact coefficients depend on the density of repulsors and the desired driving behavior).

One problem with the ODA is that it can react greatly to the presence of an obstacle that is relatively far away. Such an obstacle does not make much of a projection onto the sensors and thus can not affect the neural networks as immediately. The fix for this is to use the IDOA, or Improved Omniscient Driving Algorithm. IODA is just ODA with a limit on the amount the steering direction can change in any one cycle. This limits the initial effects of an obstacle, which means that an obstacle's initial (long range) effect on steering direction is relatively smaller.

Another issue is that the IODA can fail to navigate the vehicle safely when two obstacles are closer together than the width of the vehicle. That is, if each obstacle is the same distance in front of the vehicle, with one just to left of the vehicles center and the other just to the right, each will push the vehicle towards the other. The net sum of the repulsions is that the vehicle will head straight forward and hit both obstacles. See Figure 34 on page 83. We will see later that our neural networks learn to generalize obstacles and overcome this problem.

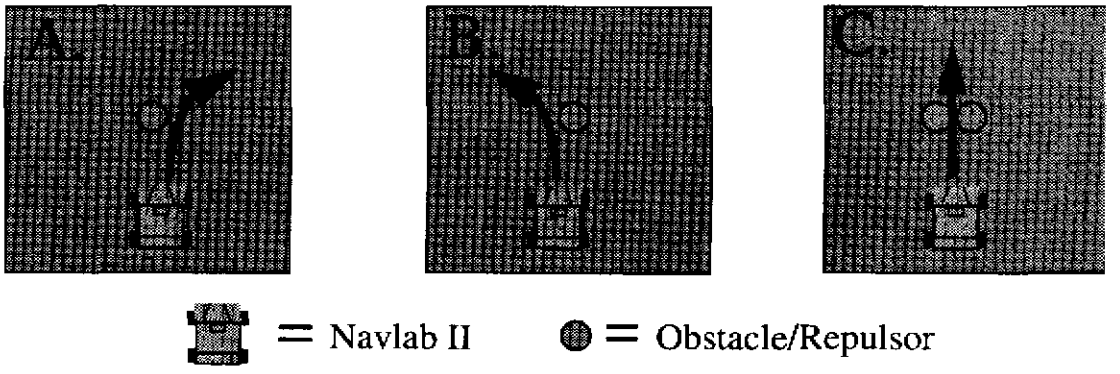


Figure 34 A) Effect of an obstacle just to the left of the vehicle's center on IODA
B) Effect of an obstacle just to the right of the vehicle's center on IODA
C) Summed effect of a both obstacles on IODA

7.2.2 *On the vehicle*

The real Navlab II presents its own issues and opportunities, though the interface to sensing and control is identical to the interface used in Malsim. We have different sense speeds. We have different computing resources available to us. We have a greater need to do things quickly as the real vehicle is constantly in motion. And, of course, we test the real vehicle in much more complicated environments than the simulated vehicle.

The system we use on the Navlab II for control is called INNNAV (Instant Neural Network NAVigator), and consists of three or four modules (see Figure 35 on page 85 for an diagram) which run in parallel on different computers and communicate with each other over the vehicle's communications network. The main modules and their functions are described below.

- Control Module. This is the main event loop for the system. When the system is in "Learn mode" this module simply lets the imaging modules know when to digitize and sends the corresponding steering direction to the IO module. When the system is in "Control Mode", this module tells the imaging modules to digitize images, waits for the images or neural representations of these images (the activations of the hidden units of some FeNNs to be described later), executes the appropriate neural network (monolithic control net, or a Task Net for a MAMMOTH Net), executes the steering actuation, and sends the steering direction to the IO module for storing or display.
- Range and Color Modules. These two modules wait until they receive commands to digitize from the Control Module. They then digitize images and reduce them to an appropriate resolution. If the robot is being controlled by a MAMMOTH network, each of these modules executes a forward propagation of activation through a single FeNN and sends the hidden unit representations to the Control Module. Otherwise, they send the low resolution images to the control module.
- IO Module. This module handles Input/Output to files and to a monitor in the vehicle. This is handled separately and asynchronously so that these potentially expensive operations do not slow down the crucial system execution.

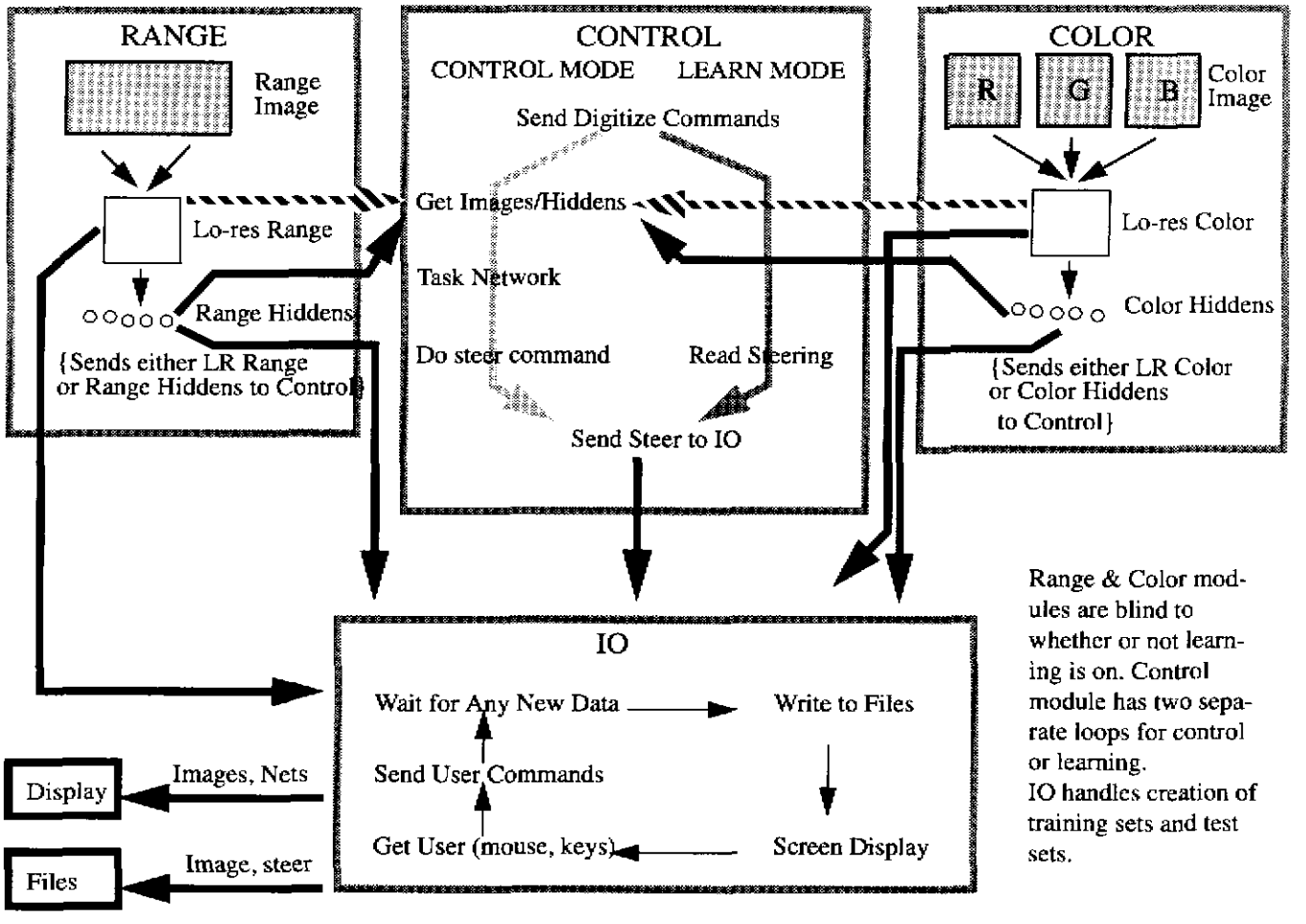


Figure 35 Block diagram of INNNAV

Ideally, each module runs on its own computer, and the whole control loop cycles as quickly as possible. The Navlab II currently has three general purpose workstations (two SPARC 10 workstations and one fast VME SPARC II) and a separate computing card for the controller. All of the computers are connected on a small and fast network.

There are two main sensors, a Sony CCD color video camera and the ERIM laser range finder. The CCD camera can generate images at frame rate (30 frames/second); data transfer can take 300 milliseconds for a high-resolution image and 50 milliseconds for a low resolution image. The ERIM can generate two frames per second (and we can get them that quickly, too). Each of the sensors feeds through an interface card into a computer dedicated to handling the low level image processing.

The control computer takes the results of the lower level image processing and generates steering commands. These commands are sent directly to the low-level controller's computer, and the steering wheel conforms to the command.

As can be seen in the diagram, we take advantage of the multiple computers by having digitizing and low-level image processing occur at the same time for both sensors. In the current system, we are tied to the speed of the slowest component. When we are using the laser range camera, the bottleneck for the system is the 2 Hz digitizing rate. Typical neural networks only take 50-100 milliseconds to execute (by execute, we mean propagate activations forward). Small Task Nets can take as few as 5-10 milliseconds to execute. When using FeNNs, communications are relatively fast as we only send hidden unit activations across the network (as opposed to images). Thus, our cycle frequency is consistently better than 1 Hz.

7.3 Navlab II Methods

For our Navlab II experiments, we once again address the basic questions we asked in the chapter "Introduction and Philosophy".

Should we use an image local or image global technique? For our Navlab II tasks, we are concerned with image features that cover large areas, such as mounds of dirt or large patches of grass. The features we are trying to seek/avoid in our particular tasks can be large relative to the size of the images, so we use an image global technique. We are concerned with the steering direction implied by the global features of the image (such as large obstacles or environmental features).

What resolution is appropriate? For a cross country navigation system to be truly useful, we would need to be able to sense and avoid obstacles of any size that would be dangerous to the vehicle. There are obstacles which present difficulties to a robot system at almost any resolution (such as a steel spike pointing towards the vehicle which would be hard to see from straight on). For our purposes, we are demonstrating basic capabilities, and we are only concerned with very large obstacles. We use laser range images as well as color video images, and our initial heuristic was to keep the number of inputs on the same order of magnitude as those used for ALVINN [Pomerleau91]. ALVINN had a 32 x 30 pixel grayscale input, for 960 inputs. We use a 16 x 64 range image and three 15 x 20 color images as inputs, for a total of 1924 inputs. With the current system, we can process the monolithic network in about 60 milliseconds on the real Navlab II (the monolithic neural network is slower than the MAMMOTH networks whose FeNNs can be executed in parallel). Doubling the resolution in both x and y for all of the images should multiply that time by four, bringing us to almost a quarter of a second just for the neural network processing, which could still be reasonable. Doubling the resolution again would bring us to one second for the neural network processing alone. Note that the training time also increases faster than linearly with an increase in the number of connections (such as would happen by using higher resolution images).

What is our training signal? Our training signal is the images generated by our sensors and the corresponding steering directions provided by the trainer (either the human driver or the IODA, Improved Omniscient Driving Algorithm).

How do we perform functional decomposition? Our features are combinations of range and color features (for example, grass mounds vs. dirt mounds). When we use MAMMOTH networks for our Navlab II tasks, we decompose the function along the boundaries of the sensor spaces. That is, one subfunction (FeNN) finds range features and one finds color features. One of the main reasons for this decomposition is that the training signal for the subfunctions are easily available (we can always limit ourselves to looking at just one sensing modality and associating just those images with a steering signal). Another practical reason on our current system is that each sensor can be read from a separate computer and the FeNNs features computed in parallel.

What are our Virtual Sensors? We can think of the Task Net of a MAMMOTH network as a monolithic network trained and executed on data from Virtual Sensors. In this case, our virtual sensors sense range-features-related-to-navigation and color-features-related-to-navigation.

The Navlab II control algorithm is a very simple reactive system shown below.

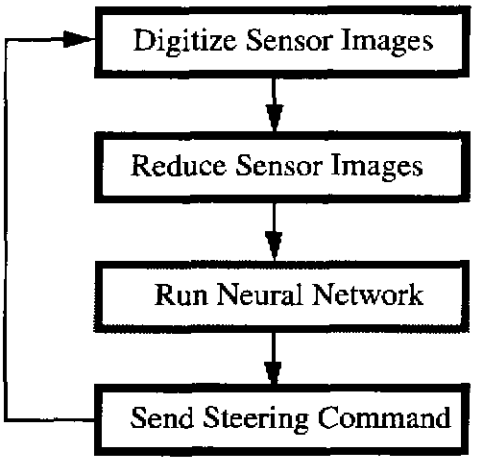


Figure 36 Navlab II Control Algorithm

The bulk of the processing is performed in the mapping from input images to steering command (the third box down in Figure 36 on page 88), and the module that performs that function is the only thing that changes from one experiment to the next.

7.3.1 *Monolithic networks*

We use two slightly different monolithic neural network architectures in our Navlab II experiments. Most of the time we used a simple three layer feedforward network with inputs from the ERIM and the CCD camera, several hidden units and eleven output units. The eleven outputs were chosen to be the same number of turning options that Kelly's Ranger system [Kelly93] has. The best number of hidden units varied depending on the specific task and training data, and were determined either automatically by ANGEL or by searching through likely candidates in a non-automated fashion. Our list of likely candidates for non-automatic searches frequently started around four or five hidden units and would go as high as a dozen. This short list was based on results from Pomerleau [Pomerleau92] and from experimentation.

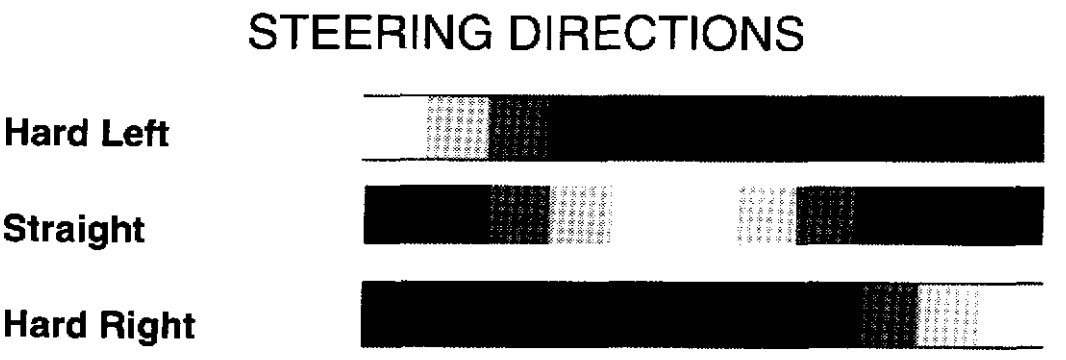


Figure 37 Output activation pattern

The output activations were done in a gaussian form, such as Pomerleau used, so that the activation was highest on node corresponding to the desired steering direction and slope off on the nodes to either side (in a gaussian shape; see "Output activation pattern" on page 89). On a few occasions an additional hidden unit of five nodes was added between the first hidden layer and the outputs in order to make the number and complexity of connections similar to those found in the MAMMOTH network used (typically, we did not observe this influencing performance in a measurable and consistent way). See "Monolithic Navlab II Multi-sensor Neural Network" on page 90 for a view of this network.

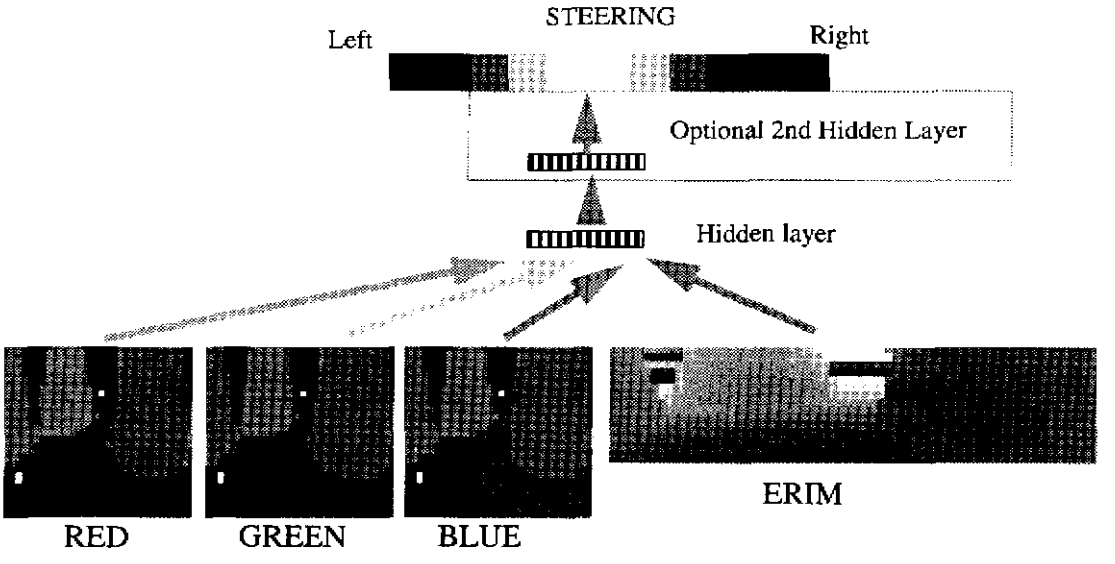


Figure 38 Monolithic Navlab II Multi-sensor Neural Network

The monolithic Navlab II networks have four input groups (red, green, blue, and range images), which constitute a single input layer from a training and execution standpoint. Which is to say that each node of the first hidden layer connects to every node of all of the inputs. Although it is convenient to express the network as in Figure 38 on page 90 where the input units are grouped by sensor, the input nodes could be in any order, and mathematically, nothing would change for the monolithic network.

To train our monolithic networks we use a simplified version of the ALVINN training set construction. We simply drive and record sensor images and their corresponding steering directions. The specifics for each task are covered in the "Experimenting" chapter, but in general the only massaging of the data set is in making sure the numbers of turns are balanced and the occurrence of pertinent features is balanced. For example, for a road following network, the pertinent features might include examples of staying on the road and examples of returning to the road after a slight error.

No artificial images are generated from the actual images to represent slight shifts in position as Pomerleau has done. This is because, for any task with obstacle avoidance, even a very small change in position could make us want to avoid the obstacle on a different side. The straightfor-

ward road following task is nice in that we can compute shifted images and the shifted steering direction is obvious, since there is *only one right place to go*. Such is not the case for obstacle avoidance.

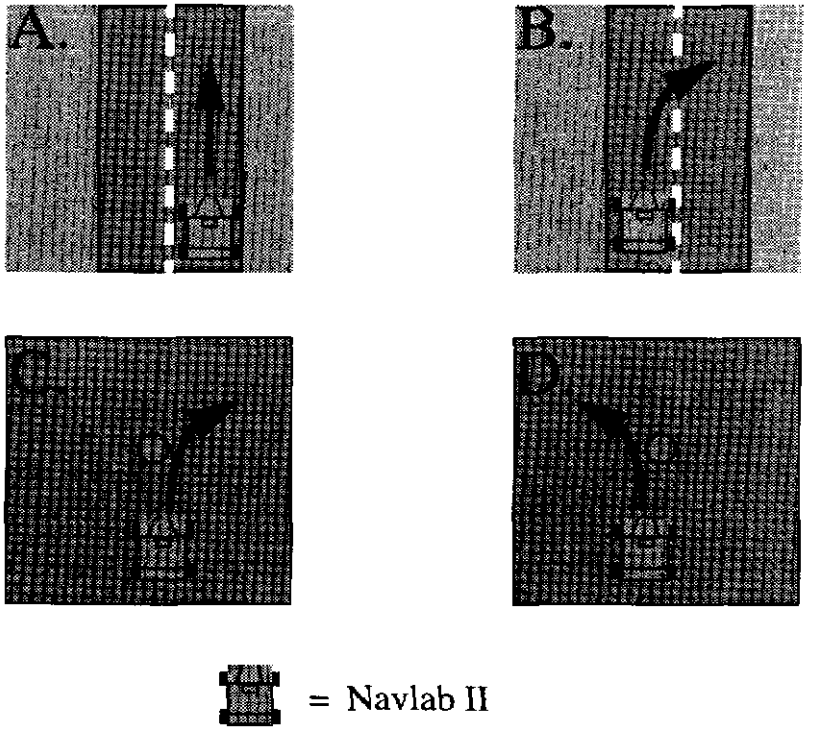


Figure 39 Why We Can't Shift Images To Generate Larger Training Sets (view from top)

Figure 39 on page 91 illustrates this difficulty. Let us consider frame A to be the position of the vehicle when an image and steering direction are recorded for training a road following system. We can create a shifted image from the original in which the vehicle is now at the position shown in frame B, and the steering direction shifts simply to that shown by the arrow, which points us to the same place. Frame C shows the position of the Navlab II when recording an image and steering direction for avoiding an obstacle represented by the circle. If we shift the vehicle's perspective just a small amount, we no longer can just shift the steering direction to head towards the same place. We cannot actually compute the new correct steering direction without knowing where the obstacles are, which we don't know (if we did, we would not need the neural network). The only case in which we could generate artificial images was in training range FeNNs, feature networks whose

only inputs were range data; the center of the ERIM rangefinder points straight out along the longitudinal axis of the Navlab II, so we generated a mirror image for each range image (cutting data collection time in half).

7.3.2 *MAMMOTH*

The next step in our methodology has us trying modular networks in our Navlab II control loop. As mentioned earlier, our division into FeNNs is by sense modality. This offers two practical benefits: easy training signal construction and parallel processing of FeNNs. Although how each network is trained varies by the specific task, for this work we use the same two FeNNs and Task Net for all of the problems (although the exact dimensions of the inputs change due to changing vehicle configurations). In some cases, we even reuse trained FeNNs.

7.3.2.1 RGB Navlab II FeNN

The FeNN we use to find features in the RGB CCD images is almost an ALVINN-network. The primary difference is that we have three color input groups instead of a single grayscale input group; but from a learning and computational standpoint, the three groups only form one layer. For our cross country navigation tasks, the color data is needed for handling vegetation, and for our simulated road navigation tasks we use obstacles of differing colors. Training is done as with the monolithic multi-sensor Navlab II network; the specific training sets depend on the task (seek or avoid vegetation). See Figure 40 on page 93 for a diagram of this network.

7.3.2.2 Range Navlab II FeNN

The Range FeNN is identical to the RGB Navlab II FeNN (modulo sensor aspect ratio). The input layer consists just of one group of range sensor inputs. The output layer is a steering vector with a gaussian activation function.

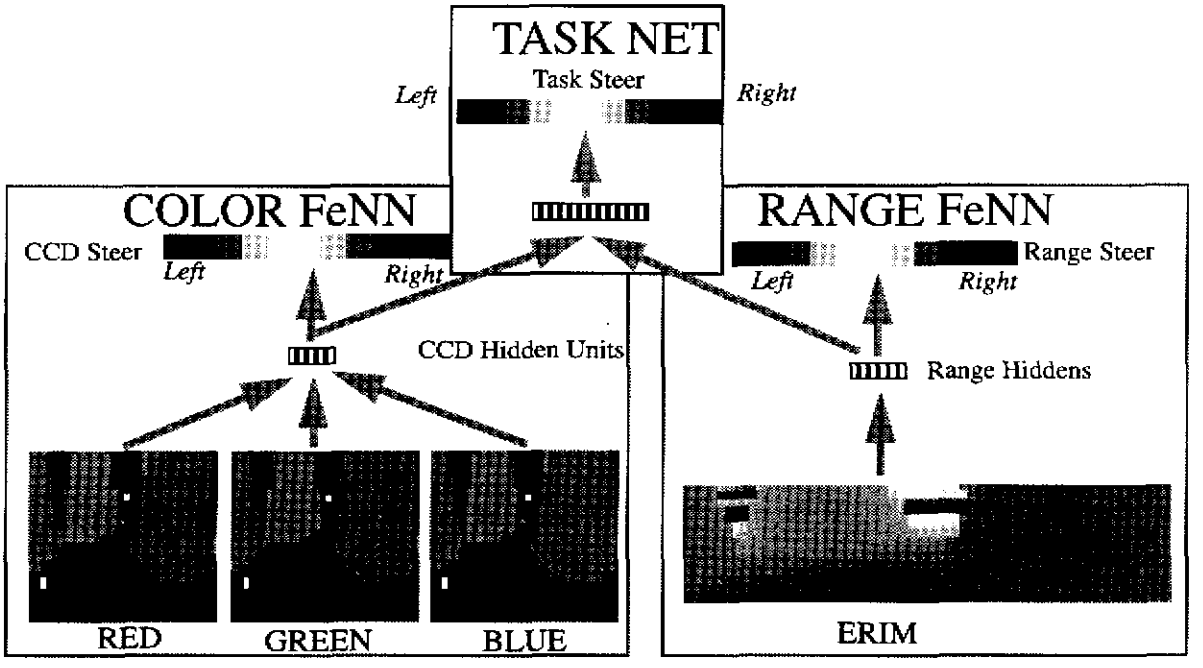


Figure 40 Navlab II Range-Color MAMMOTH

7.3.2.3 Navlab II Task Net

For our control loop, the data from the range and color FeNNs is the only data fed into the Task Net. One could see using multiple FeNNs in each sense modality in order to find different features, but for our tasks so far we use only one FeNN for each modality.

The Task Net is a simple three layer neural network whose inputs are the hidden units of the FeNNs. A training set is generated for the Task Net by driving the Navlab II and gathering images in both sensing modalities and the corresponding steering directions. The image digitization is synchronized, so we currently cycle at the rate of the slower sensor (which is typically the laser range

finder on the real vehicle). Each of the images is run through its corresponding FeNN, and the hidden units from the FeNNs and the steering directions become the exemplars that make up that actual training set for the Task Net.

7.4 Navlab II Experiments

Our Navlab II experiments have two primary goals: 1) show new XC navigation capabilities with either neural approach, and 2) show that MAMMOTH offers advantages in terms of training time. When training networks on a given task, we typically have to train a large number of networks several times in order to be sure we have the best network architecture, so training times become significant. Although there is frequently a penalty to pay in the time is takes to generate a training set, it is usually offset by the savings in training times. We will measure the time-to-train in this section the number of weight adjustments required to train a network to its peak performance. For reference, a typical fast workstation (for example a SPARC 20) using the neural network code we use can average almost 5,000,000 weight adjustments per hour (bear in mind that this code is not optimized for speed). Recall that each weight adjustment involves doing a forward pass through each exemplar in the training set, which for most of our Navlab II networks is 1000 image/steering direction pairs.

We applied our techniques to two areas of navigation, road navigation and cross country navigation. For all of our tasks we used the standard Navlab II monolithic network and the Navlab II MAMMOTH network. The output layer always had 11 outputs, and the input layer had 20x15 pixels for each or the three visual bands of the CCD image (red, green, and blue) and 64x16 pixels for

the laser range image. The numbers of hidden units varied slightly from experiment to experiment, but were typically in the ranges of 4 to 10.

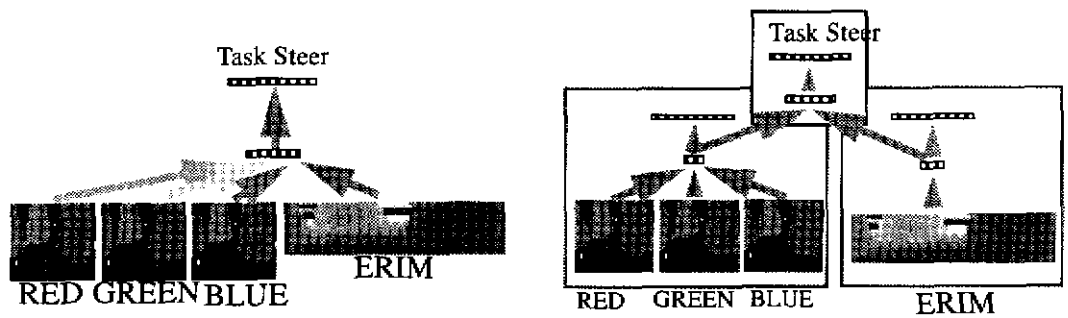


Figure 41 The Navlab II networks: Monolithic on the left & MAMMOTH on the right

7.5 *Road Following to Road Navigation*

Two experiments in Malsim take the road following work of Pomerleau a step further by adding obstacle avoidance with the laser range sensor. Pomerleau's experiments used only the video camera (in gray scale mode with shadow correction through the blue band) which is significantly faster than the 2 Hz ERIM, and this allowed him to achieve highway speeds. For our experiments, we had to slow down in our virtual world to accommodate the virtual ERIM, but we achieved a new level of capabilities.

7.5.1 *Road Following with Obstacles*

Test Site:	Simulated Road on toroidal world. Black obstacles in road.
Training Signal Input:	Simulated ERIM [16 x 64] & CCD [15 x 20 x 3 {RGB}] images
Training Signal Output:	Steering angle generated by human driving in simulator
Training Set Size:	300 Exemplars each for Monolith, FeNNs, & Task Net (with a like-sized test set)
Metrics for Success:	Distance travelled without colliding with an obstacle or leaving the road
Duration of Experiment:	4000 cycles (approximately 40 traversals of test road)

Navlab II

Networks Tested:	Standard Navlab II Monolith, Standard Navlab II MAMMOTH, hidden units chosen empirically as best (7 hiddens for Monolith, 5 for color FeNN, 7 for range FeNN, 7 for Task Net).
Result of test:	Demonstrates new Road Navigation capabilities with neural networks. Shows speed advantage of MAMMOTH network

The first experiment involved following a road while avoiding obstacles. The obstacles were stationary and black, which was the same color as the road. The vehicle had to use both sensors (color video and laser range) to accomplish this goal. Figure 42 on page 96 shows a map of the simulated terrain. There are block shaped obstacles at points A, B, & C. A black pit is at point D. The road we want the vehicle to follow is the paved road with the yellow line down the middle. There is a "dirt" road (brown) running east-west, and various geometric obstacles off-road.

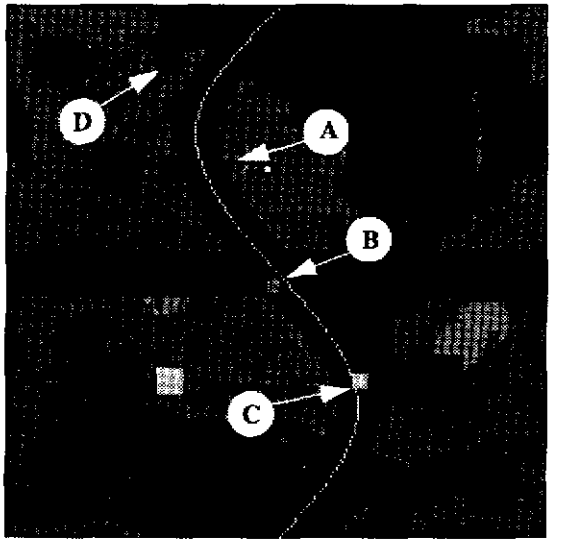


Figure 42 Map of the road-obstacle world. There are block shaped obstacles at points A, B, & C. A black pit is at point D.

Both monolithic and modular networks were used for this task. We collected training exemplars once per simulated meter. For the monolithic network and the Task Net part of the MAMMOTH network, the vehicle was driven (using the mouse to steer) over the road several times in each direction. Additional exemplars were gathered by driving off the road (with data collection off) and then driving back onto the road; these exemplars were necessary so that the simulated vehicle could

learn to recover from small navigational errors. All the while, the vehicle was steered so as not to hit any of the three obstacles on the road and to drive on the road whenever possible.

For the monolithic network we were able to train with just this data; for the MAMMOTH network, we first trained the FeNNs. One FeNN was for road-navigational features using the simulated color CCD camera and the other FeNN was for the obstacle-avoidance-navigational features using the simulated laser range finder.

Malsim gave us the power to make training sets for the FeNNs easily (for a discussion of how to train similar FeNNs in the real world, see "Real-world additive obstacle experiments" on page 100). The nature of a FeNN is that we want it to learn internal representations (weight patterns) that represent a narrow and known set of features. In Malsim we can force a FeNN to learn only what we want by creating worlds whose only features are those we wish to learn. To train a FeNN to recognize low-level video features of roads appropriate for navigation and not to key on extraneous image features, we make a world in which the only feature is a road (and its complement, the grass or dirt at the side of the road). Likewise, to train an obstacle avoidance FeNN, we make a world in which the only features are large obstacles to be avoided. See Figure 43 on page 97.

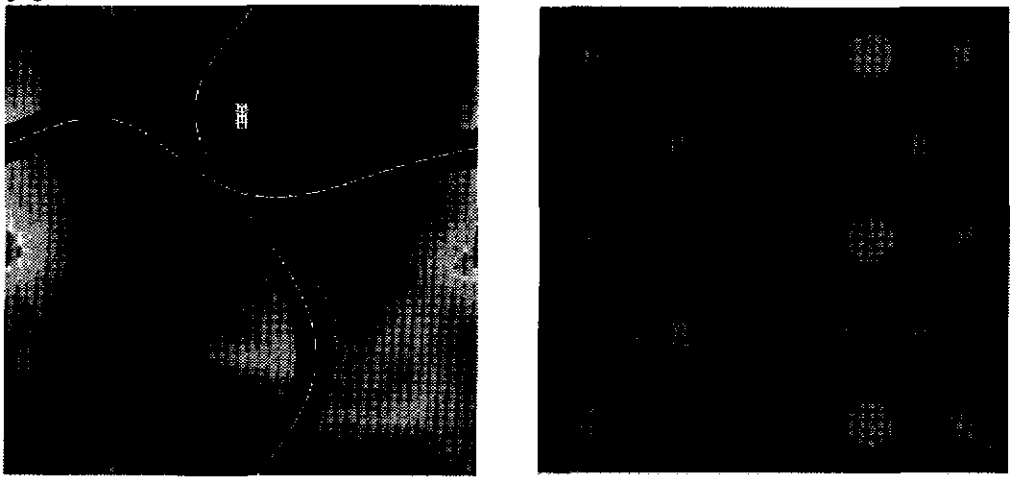


Figure 43 Road world & obstacle world

For the MAMMOTH network, 300 training exemplars were gathered in both the road-world and the obstacle-world. The respective FeNNs were trained with these and then the weights to the outputs of the FeNNs were frozen. Finally, the same training set used on the monolithic network was used to train the Task Net part of the MAMMOTH architecture.

Performance was excellent on this task for both network architectures. For both training and testing the speed of the simulated Navlab II was held constant, and the vehicle was able to stay on the road and avoid obstacles for more than 4 simulated kilometers. Another option in our Malsim simulation allowed us to "warp" the vehicle to a random location and orientation in the simulated world, and three times out of every four, the vehicle resumed travelling on the road when it intersected the road again. About every fourth time the vehicle would intersect the road at almost a right angle and that sort of example was not in the training set, so it would not make the turn onto the road.

7.5.2 *Road Following With Colored Obstacles*

Test Site:	Simulated Road on toroidal world. Brightly colored obstacles in road.
Training Signal Input:	Simulated ERIM [16 x 64] & CCD [15 x 20 x 3 [RGB]] images
Training Signal Output:	Steering angle generated by human driving in simulator
Training Set Size:	300 Exemplars each for Monolith, FeNNs, & Task Net
Metrics for Success:	Distance travelled without colliding with obstacles or leaving road
Duration of Experiment:	4000 cycles (approximately 40 traversals of test road)
Networks Tested:	Standard Navlab II Monolith, Standard Navlab II MAMMOTH, hidden units chosen empirically as best (7 hiddens for Monolith, 5 for color FeNN, 7 for range FeNN, 7 for Task Net).
Result of test:	Demonstrates new Road Navigation capabilities with neural networks, Shows speed advantage of MAMMOTH network Hints (and only hints) at advantage of having narrowly defined FeNNs

Another simulator task highlights one of the pitfalls of using neural networks and the results suggest two solutions. The goal was again to stay on the road and to avoid the static obstacles also on the road, but this time the three obstacles were colored bright blue, green, and yellow from north to south along the road. Training sets were gathered in exactly the same manner as in the task of Section 5.1.1 on page 11, and training was done for the same amount of time (approximately 600 epochs - or times through the whole training set in batches of 300 - for each network to train).

The results were similar to those of the previous section. The particular training set for this task produced slightly sloppier driving habits in the simulated Navlab II, and this was evidence by the occasional (approximately 1 in 8 occurrence) of driving fairly widely around the blue box at point A and then heading straight into the pit at point D (refer again to Figure 42 on page 96). This behavior was bad. However the explanation and fix are simple. The reason this happened is because the vehicle had not driven so far to the left on the road near point A before and thus had not seen the pit very well before. In this simulated world, the pit and the walls of the pit are black - the same color as the road. As the road turned back to the right, if the vehicle had just avoided the block at point A widely, the vehicle would see the black of the pit better than the black of the road and head that way. The fix is simply to add examples of avoiding pits to the training set.

A more interesting result came up when these networks trained with the blue, green, and yellow blocks were tested on a simulated world which was identical except for having three bright red blocks instead of three differently colored blocks (with the blocks at exactly the same locations). The first symptom was that the vehicle went off the road and into the pit much more frequently. With the monolithic network there was at least 2 in 3 chance of going off of the road, and with the modular network there was approximately a 1 in 3 chance. The implication is clear. The person training the network had correctly driven more closely around the blue box at point A than the yellow box at point C in the training. And both networks had learned to go left around blue boxes more tightly than around yellow boxes. With the red blocks, the network drove much more loosely around the block at point A, and saw the pit that much more often.

The second symptom was that the vehicle frequently went off of the road at point B. There was an almost 100% occurrence with the monolithic network and at least a 50% chance with the modular network. Both of these symptoms have the same explanation: the yellow block in the first colored-block world was the only block with a red component in RGB. The network learned to go wide around the yellow block since it had to go around it on the left to stay on the road and there was also a left turn immediately after the block on the road. This made the network go widely to the left of the block at point A and often to the left of the block at point B.

A simple fix is to use bigger training sets which have examples of blocks of every color on both sides of the road. But it is also interesting to note the slightly better performance of the modular MAMMOTH network. The training set for the Task Net was identical as that for the monolithic network, but it does seem to have helped that the lower level representations were separated into representations strictly for obstacles in laser range space and representations strictly for road following. By designing our FeNNs to narrowly represent what we want, we seem to have decreased the amount that we keyed on incidental features (our research to date supports this only anecdotally; however, this is a matter worthy of future study).

7.5.2.1 Training times

The training times for all of the networks involved were between 300 and 600 epochs. Several versions of each network (monolithic, FeNNs, & MAMMOTH) were trained (starting from different random weights), and typical training times were 400 epochs for the color FeNN, 500 for the monolithic network and MAMMOTH network, and 600 epochs for the range FeNN (there were many different sized and shaped obstacles in the range FeNN training set, which we believe made the training time longer). Therefore, the monolithic network with 1924 inputs, 7 hidden units, and 11 output units required 6,772,500 weight adjustments (13545 connections times 500 epochs). Since the FeNNs can be trained in parallel, we need only consider the larger, slower FeNN's (the range FeNN's) training time of 4,347,000 weight adjustments (7245 connections times 600 epochs); the Task Net would add only 80,500 weight adjustments (161 connections times 500

epochs). The MAMMOTH networks took a total of 4,427,500 weight adjustments to reach peak performance on a test set, or 65% as long as the monolithic network. Recall that the training times and number of hidden units were determined by ANGEL (see "Completing Training" on page 14).

7.6 Cross Country

Our last experiments involve two tasks which demonstrate the need for multiple sensor modalities in XC navigation. The first task involves avoiding two types of obstacles, obstacles that appear only in the color sensor space and obstacles that appear only in the range sensor space. The second task involves using both sensor modalities to define the single type of obstacle that needs to be avoided.

7.6.1 Real-world additive obstacle experiments

Test Site:	Real Slag Heap, Site 1 [3km long access road with open areas]
Training Signal Input:	Real ERIM [16 x 64] & CCD [15 x 20 x 3 (RGB)] images
Training Signal Output:	Steering angle generated by human driver [11 choices]
Training Set Size:	1000 Exemplars each for Monolith, FeNNs, & Task Net
Metric for Success:	Distance travelled until human safety driver declared situation unsafe due to impassable obstacles or culs-de-sac
Duration of Experiment:	25 runs for each architecture.
Networks Tested:	Standard Navlab II Monolith, Standard Navlab II MAMMOTH, number of hidden units chosen empirically (7 hidden units for monolithic network, 7 for color FeNN, 7 for obstacle FeNN, and 7 for Task Net)
Result of test:	Demonstrates new XC capabilities (handles grass and obstacles), Shows speed advantage of MAMMOTH network

If everything in the world was a rigid body, we would know to simply avoid everything, and we could do it [Kelly93]. However, sometimes we want to drive over the vegetation when we have no other decent route. The laser range finder alone cannot sense the low vegetation such as grass, so we must use an additional sensor, in this case the color video camera to sense such low vegetation.

The first task has a simple goal: don't hit big solid obstacles and also don't hit vegetation. We call the first task the additive obstacle task because the additional sensor only adds new things to avoid (grass and low vegetation). We trained and tested in an area of the slag heap containing an access road which often had vegetation and/or earth mounds on one or both sides.

We used the same network architectures as for the road following tasks. It is clear, however, that generating appropriate training sets is significantly more difficult in the real world than in the simulator. For the Task Net part of the MAMMOTH network and for the monolithic network, we had to drive the Navlab II around the terrain. This would have been simple, except we still needed examples of recovering from mistakes, which meant we had to make mistakes (with data collection off), and mistakes are harder to make in a five ton truck than in a simulated five ton truck. The procedure we used was the same (drive off of the road or too close to a mound or cliff and then recover to the right position), but the implementation was more difficult. Consequently, speed varied greatly, and the training set was slightly less rich in recovery examples than the training sets in the simulator.

7.6.1.1 Navlab II Monolithic Network

Gathering data for the monolithic net (and for the Task Net) took about an hour on average. The training set was balanced with a 50/50 mix of straight (or near straight) steering directions and hard turns. Among the turns, half were left and half were right. We usually gathered 600 training exemplars. Although this corresponds to only ten minutes of straight driving (gathering data at the maximum speed of the slower sensor), we had to provide examples of recovery from mistakes, so we frequently had to stop and maneuver the vehicle into a position that would allow us to "recover" correctly so that the networks could learn to recover from errors.

7.6.1.2 Obstacle FeNN

Training the FeNNs was the real challenge, though. In the real world we cannot simply train in a world that contains the narrow set of features we wish to detect (most of the time). The laser range FeNN was trained to avoid obstacles, as it had been in the simulator. There were several large mounds of earth sufficiently far from other objects (including vegetation) which we used as our prototypical obstacles, and we trained simply by avoiding these mounds as we had done in the simulator. Since these mounds were conveniently isolated, gathering 600 images usually only took about half of an hour (the set was balanced with straights/turns and lefts/rights as above).

7.6.1.3 Vegetation FeNN

We also wanted the CCD video camera FeNN to produce representations of vegetation, and this was more difficult. We trained a FeNN to drive around avoiding vegetation. In the real world, though, images containing vegetation often contain other things; additionally, vegetation in the real world is not nearly as homogeneous as any feature we might want to recognize in the simulated world. We had to try to develop a training set that contained a rich set of examples of the Navlab II steering away from all of the types of vegetation we might encounter. This usually took more than an hour and fifteen minutes. Again, the training set was balanced for straights/turns and lefts/rights.

7.6.1.4 Monolithic Net & Task Net

Given these difficulties, the FeNNs we trained both worked quite well: each FeNN successfully avoided all of the obstacles visible in its sensor modality within the resolution of the networks' inputs. The monolithic network and the Task Net performed even better: by combining the two sensor modalities with either the monolithic network or the MAMMOTH network the vehicle would avoid both range and color obstacles within the resolution of the networks' inputs in. In fact, our two main problems were that in the real world there are plenty of obstacles that simply don't show up at the resolutions we used in sensor space (such as sharp steel bars and jagged cinder blocks) and that there are plenty of culs-de-sac at the slag heap.

Both of the FeNNs would quite frequently allow the Navlab II to drive for runs of up to 0.6 miles before we ended up in a cul-de-sac or had something dangerous right in front of us that could not be seen at the low resolutions at which we used the sensors. Another failure mode was usually the result of uneven ground or territory not covered in the training sets for the FeNNs producing images that were largely dissimilar to all of those in the training sets for the respective FeNNS. The most frequent failure mode for the CCD FeNN was driving right up to a cliff that was the same color as the road. Likewise a common failure mode for the laser range FeNN was driving into low vegetation that got progressively more dense until we gradually appeared in an area where the vegetation was dense enough that no correct steering direction could be determined simply from range images.

Combining the two sensor modalities had the desired results. Typical runs of the Navlab II were around 1 mile or more before we encountered a cul-de-sac, a dangerous obstacle below sensor resolution or a previously unseen situation (such as a tilted image from a tilted vehicle); in 25 runs, we averaged over 1.1 miles with both (a tiny bit higher with the monolithic network, but not significantly). There was very little difference between the performance of the monolithic Navlab II network and the performance of the MAMMOTH Navlab II network on this task. Our original intention was to return to this site and perform many more iterations of this test, but hardware and policy issues prevented us from returning. Thus, this analysis must unfortunately remain largely qualitative.

7.6.1.5 Training times

As these runs were all performed out in the field, and time was of the essence, we did not get to find the *best* training times and architecture for each net (we could evaluate that off-line afterwards, but we would not subsequently be able to test in the same conditions). From several trials, we determined that 500 epochs each for the FeNNs and 750 epochs for the monolithic network and the MAMMOTH network gave us networks which qualitatively performed as well as ones trained for longer (in terms of distance travelled), and better than those trained for less time. These numbers

(which do not necessarily reflect the optimal training times, unfortunately) result in 10,587,500 weight adjustments for the monolithic network and 3,743,250 weight adjustments for the MAMMOTH network (3,622,500 weight adjustments for the slower FeNN and 120,750 for the Task Net). Recall that each weight adjustment involves performing forward passes through each exemplar in the training set.

7.6.2 *Conflicting obstacle test*

Our final test is getting the vehicle to avoid solid obstacles (dirt mounds) while allowing it to run over similarly sized compressible obstacles (bushes). The location of the real world test is a section of roughly 65 by 65 meters on the slag heap which has numerous man-made mounds of earth spaced just a bit wider than is necessary for the Navlab II to drive through. The mounds of earth are about a meter high and two meters in diameter, and between them vegetation grows just as high in spots. Our task was to drive the Navlab II through this obstacle course. The laser range sensor generated images in which the vegetation looked like mounds of earth, so the video camera was needed to differentiate between safe and unsafe steering directions. However, we needed the laser range camera for avoiding many of the mounds since they were the same color as the ground when the vegetation was sparser. The network architectures used were the same as for the road following tasks and the additive obstacle task. We call this task the conflicting obstacle task, since the CCD features conflict with what we would normally call obstacles in the range sensor space.

7.6.2.1 Real Slag Heap Site 2 Conflicting Obstacle Test

Test Site:	Real Slag Heap, Site 2 [65 x 65m, with 24 mounds of earth]
Training Signal Input:	Real ERIM [16 x 64] & CCD [15 x 20 x 3 {RGB}] images
Training Signal Output:	Steering angle generated by human driver [11 choices]
Training Set Size:	1000 Exemplars each for Monolith, FeNNs, & Task Net
Metric for Success:	Percentage of successful passes through test site
Duration of Experiment:	25 runs through Site 2
Networks Tested:	Standard Navlab II Monolith, Standard Navlab II MAMMOTH

On the real Slag Heap, results have been slower in coming for the conflicting obstacle test due to a number of engineering and political issues, but some preliminary tests have been run. Our testing used the same network architectures as in the simulator and as for the real world additive obstacle task. Our minor results were gathered over two testing days (separated by a year). For the first day, we gathered 600 color FeNN exemplars and 600 monolithic/Task Net exemplars; we actually reused the range FeNN from the additive obstacle test (we could not reuse the color FeNN since the vegetation at Site 2 was very different in color from that at Site 1; at Site 2 it was tall, brownish

weeds and brush, whereas much of the vegetation at Site 1 was low green grass). For the second day, we gathered 1000 exemplars for each of the FeNNs and for the monolithic and Task Net.

In the few outings we've had, the monolithic network has failed to perform at all, and the modular network has been successful about 44% of the time (we got through the test site 11 out of 25 times). One cause of failure is the extraordinary time it takes to generate the training sets; this causes the testing to be done in different lighting situations than the initial data gathering. Another cause was that at the very slow speeds we had to use, the vehicle's front-mounted sensors would often clear an obstacle before the wheels did and the purely reactive system would no longer see that obstacle, and the vehicle would sometimes turn into that obstacle to avoid the next one.

Training times and data gathering times were similar to those in "Real-world additive obstacle experiments" on page 100, but the results were poor, so these measures do not mean much for our purposes. We had some minor success avoiding obstacles with the re-used range FeNN, but not enough to be significant.

7.6.2.2 Simulated Slag Heap Site 2: Two Conflicting Obstacle Experiments

Although repeated hardware failures and policy issues prevented us from spending any more time at the real Slag Heap, we had a rich simulator in which to conduct additional tests. We were able to demonstrate the basic Cross Country Conflicting Obstacle capabilities in Malsim through two closely related experiments. We roughly recreated Site 2's layout of mounds of dirt, and scattered an equal number of vegetation mounds throughout the test site. In the simulator, the only difference between the dirt and vegetation mounds was the coloring. The dirt mounds had the same color as the dirt ground (shades of brown) and the vegetation mounds were shades of green.

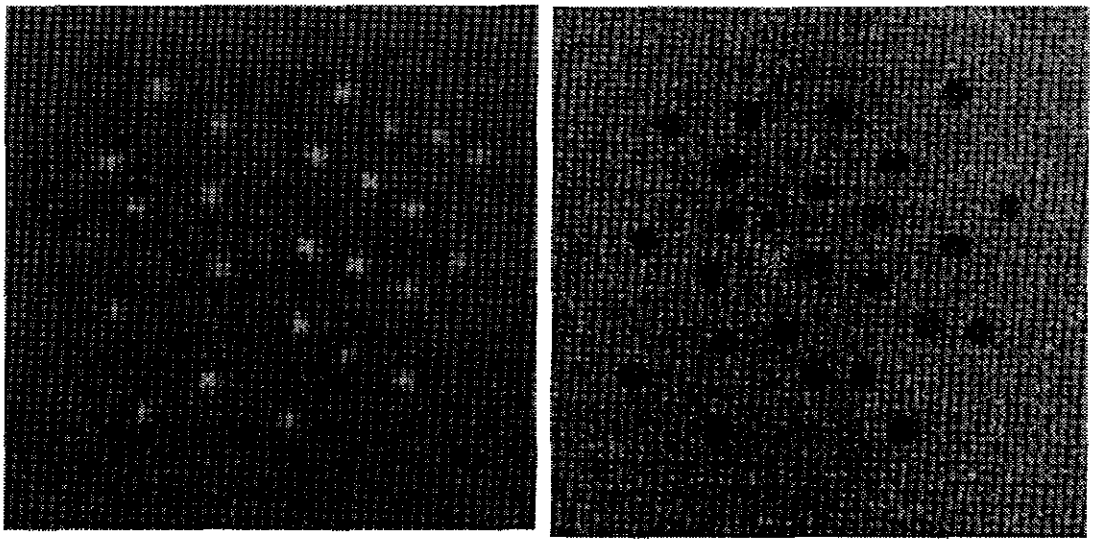


Figure 44 Simulated Slag Heap Site 2 elevation map and color map

7.6.2.3 Simulated Slag Heap Conflicting Obstacle Experiment #1

Test Site:	Simulated Slag Heap, Site 2 [100 x 100m, with 65 x 65m section containing 24 mounds of earth]
Training Signal Input:	Simulated ERIM [16 x 64] & CCD [15 x 20 x 3 {RGB}] images
Training Signal Output:	Steering angle generated by IODA [11 choices]
Training Set Size:	1000 Exemplars each for Monolith, FeNNs, & Task Net
Metrics for Success:	Deviation % of neural network steering from IODA steering
Duration of Experiment:	4000 cycles (approximately 40 traversals of test site), repeated 3 times
Networks Tested:	Standard Navlab II Monolith, Standard Navlab II MAMMOTH, hidden units chosen by ANGEL
Result of test:	Demonstrates new XC capabilities (handles grass and obstacles in conflicting test), Shows speed advantage of MAMMOTH network Shows networks ability to generalize obstacles over IODA

The first simulated Site 2 experiment used the Improved Omniscient Driving Algorithm (see "Omniscient Driving Algorithm (ODA) & Improved Omniscient Driving Algorithm (IODA)" on page 82) to generate training signals for networks on the simulated slag heap; all of the testing was done in the same simulated slag heap. The goal of the monolithic network and the MAMMOTH network in this task is to avoid dirt mounds, but run over (ignore) vegetation mounds (note that ignoring vegetation mounds means that the network can distinguish vegetation mounds from dirt mounds, but it does not avoid the vegetation mounds). Our main metric of success is how well the networks learned to mimic the IODA, and this is reflected by the deviation from the IODA's predicted steering direction at any given point in time (expressed as a percent of the total range of possible steering directions; thus, if the IODA said, "Steer hard left," and a network said, "Steer hard right," this would represent a 100% deviation).

7.6.2.4 Data Collection and Training for the Monolithic Network

We generated the monolithic network's training set for the simulated slag heap with the Omniscient Driving Algorithm. For this training set, the IODA only considered the dirt mounds to be obstacles, and would drive straight through the vegetation mounds if they lay in the path. Furthermore, due to the set up of the simulated test site, the vegetation mounds were run over frequently. We gathered 1000 training exemplars. Half of the exemplars corresponded to driving straight ahead and half to turning. These 1000 exemplars constituted the training set. Another 1000 exemplars gathered (from a different starting point) for a test set.

Once we gathered the 1000 exemplars of the training set, we trained our monolithic network for the conflicting obstacle task. We used ANGEL to find the correct number of hidden units for the task. For this task, five hidden units was best. 2150 epochs provided optimal performance on the test set.

7.6.2.5 Data Collection For The FeNNs

Generating training sets was easy using our Improved Omniscient Driving Algorithm. For the range FeNN, we simply created a world that was full of dirt obstacles each repulsed the vehicle. The mounds were far enough apart that the Navlab II could always fit between two of them. This let us generate a FeNN with features relevant to navigating amidst solid mounds. We generated 1000 training exemplars, and when training was finished (in around 300 epochs), the vehicle could drive in this world for many kilometers (observed for over 4km several times). The best number of hidden units found was 7.

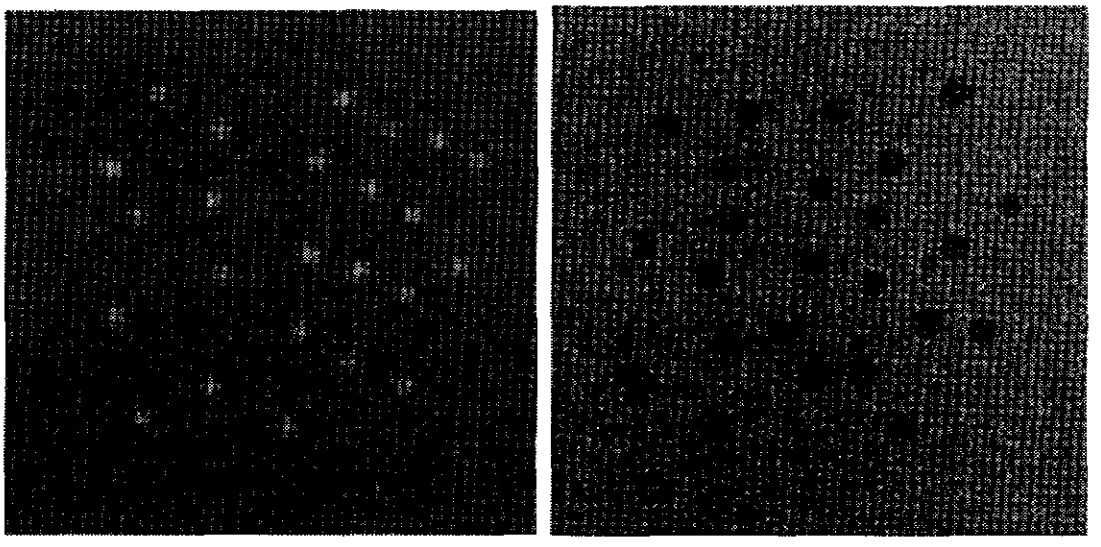


Figure 45 Simulator's FeNN training worlds: dirt mounds, and vegetation mounds

For the color CCD FeNN, we did an unusual thing. Although the FeNN would be used to identify obstacles that *could* be run over (vegetation mounds), we trained the FeNN to *avoid* vegetation mounds. The color CCD FeNN was trained in a world just like the one used to train the range FeNN, except the mounds were green. In the process of learning to avoid the vegetation mounds, the FeNN generated useful features for identifying these mounds for other control purposes. The initial reason for this seeming reversal was the simple practical fact that we already had an IODA that could handle this sort of training. The secondary reason was just to show that the FeNN's internal representations developed into general vegetation features (related to navigation), to try to confirm and build upon Waibel's claim that, "*Networks trained to perform a smaller task may not produce outputs that are useful for solving a more complex task, but the knowledge and internal*

abstractions developed in the process may indeed be valuable." In this case the network with the smaller task (the FeNN) actually produces the opposite output from what the more complex task wants. Note that the Task Net was still trained to run over vegetation.

Again, we used 1000 training exemplars and observed the vehicle drive safely around the vegetation mounds for more than 4km each time. ANGEL reported 5 hidden units and 250 epochs as best for the CCD FeNN.

7.6.2.6 Data Collection and training for the MAMMOTH Task Net

The Task Net was trained on *the same raw data* as the monolithic neural networks. Note, however, that to reach their best performance on the test sets, the MAMMOTH FeNNS were trained for 250 epochs each and the Task Net for 500 Epochs. After these amounts of time, each network had achieved its peak performance on the test set (which was the same size as, but distinct from the training set). The Task Net had 10 hidden units (as reported by ANGEL).

7.6.2.7 Task Performance

The performance is measured over 4 km of driving on the simulated Slag Heap. This represents up to 40 traverses of the test site (the simulated world is 100 meters by 100 meters, although the obstacles are mostly in the inner 65 by 65 meters, and the world is toroidal, meaning if you drive off the top of the world you appear at the bottom {and vice versa}, and if you drive off the right side, you appear at the left side {and vice versa}). All of the networks avoided the correct obstacles most of the time, so the interesting measure was how closely the networks learned to mimic the ODA steering algorithm which generated the training sets.

The error metric used is the average steering deviation from the direction the IODA would choose. One source of error was that the IODA would frequently changed steering directions quickly, while the interpolating ability of the neural network techniques tended to smooth out the curves. Additionally, the IODA would be affected by an obstacle as soon as it reached an arbitrary maximum range, so obstacles had a large influence faster than with the generalizing neural networks.

When we used both the monolithic network and the MAMMOTH network trained to their peak performance (according to ANGEL), the monolithic network achieved a 6.5% deviation from the IODA, while the MAMMOTH network achieved a comparable 6.7%. Thus, we can not claim any performance improvement of MAMMOTH over a monolithic network for this task. However, the MAMMOTH network was trained more rapidly. It took fewer total epochs to achieve comparable performance (1000 to 2150). Moreover, the FeNNs can be trained at the same time, which drops the training time to 750 epochs.

Now let's consider the number of weights to be adjusted in each training phase. The monolithic network has 1924 input units feeding into 5 hiddens feeding into 11 outputs, for a total of 9675 weights to be adjusted in each of 2150 epochs. That makes a total of over 20 million weight adjustments. For MAMMOTH, the slower FeNN was the range FeNN, which had 1024 inputs feeding into 7 hiddens into 11 outputs, for a total of 7245 weights to be adjusted. In 250 epochs, this means there are just over 1.8 million weights to be adjusted. The Task Net has 12 inputs (the activations of the five hidden units from the color FeNN and the 7 hidden units from the range FeNN) feeding into 10 hidden units into 11 output units, for a total of 230 weights to be adjusted. In 500 epochs, this makes 115,000 weight adjustments. Thus, the total sequential time to train the MAMMOTH network in terms of weight adjustments is under 1.85 million adjustments, compared to 20 million for the monolithic neural network.

Consider also the hypothetical case in which the FeNNs and the Task Net all took the same amount of time to reach peak performance as the monolithic neural network. Assume each FeNN has 7 hidden units, the Task Net has 10 hidden units, and the monolithic network has 7. Assume each network takes 1000 epochs to train to peak performance on a test set. Then the monolithic network would take 13,545,000 weight adjustments until it reached peak performance (13,545 weights times 1000 epochs). Since the FeNNs could be trained at the same time, the sequential training time for the MAMMOTH network would be the time to train the range FeNN (which has a larger input) plus the time to train the Task Net, which is 7,245,000+250,000 = 7,495,000 weight adjustments.

As an experiment, we retested the same monolithic neural network that produced the 6.5% error measure, but we used the weights stored from 1000 epochs (which is still significantly more time in terms of weight adjustments than it took to train the MAMMOTH network), and the result was an 8.9% steering deviation from that of the IODA.

7.6.2.8 Generalizing Obstacles

One unanticipated result from this experiment comes from the fact that we did not train the networks in regions with obstacles too close together for the simulated Navlab II to fit between them. This had been done partly because the IODA could collide with these obstacles (see "Omniscient Driving Algorithm (ODA) & Improved Omniscient Driving Algorithm (IODA)" on page 82). However, we did 20 tests comparing the neural networks to the IODA when they were headed right at two obstacles directly next to each other in the simulator. The IODA hit the obstacles if they were lined up correctly, but the neural networks always turned away from the obstacles. The neural networks benefitted from *not having the omniscience* to know that the obstacles were really two separate obstacles; the neural networks treated them as one large obstacle.

7.6.2.9 Simulated Slag Heap Conflicting Obstacle Experiment #2

Test Site:	Simulated Slag Heap, Site 2 [100 x 100m, with 65 x 65m section containing 24 mounds of earth] Also, similar obstacle courses with randomly generated mounds [random placement of a random number of mounds {20 to 50 mounds}]
Training Signal Input:	Simulated ERIM [16 x 64] & CCD [15 x 20 x 3 {RGB}] images
Training Signal Output:	Steering angle generated by IODA [11 choices]
Training Set Size:	1000 Exemplars each for Monolith, FeNNs, & Task Net
Metrics for Success:	Deviation % of neural network steering from ODA steering Average minimum distance from obstacles Number of collisions with obstacles
Duration of Experiment:	1,000,000 cycles (approximately 10,000 traversals of test site)
Networks Tested:	Standard Navlab II Monolith, Standard Navlab II MAMMOTH, hidden units chosen by ANGEL

The second simulated conflicting obstacle experiment was similar to the first one in that it used the IODA to generate training signals for the monolithic and MAMMOTH networks. However, instead of using just the simulated Site 2, we trained and tested on a much more varied terrain, and we ran the experiment for 1 million cycles (1000km simulated run).

The essential difference is this: we changed terrain layouts frequently in both training and testing situations. Every 100-500 cycles (randomly determined), we would choose a new terrain layout. The initial Slag Heap Site 2 layout had 24 mounds of dirt. For each new layout, a random numbers of mounds between 20 and 50 was chosen). Each mound was placed randomly in the 100m x 100m test site, with the constraint that there was enough room for the vehicle to pass between the mounds (simply maintaining consistency with the real world test site).

One effect of this larger test is that there were more collisions with obstacles; many of these came from a dirt obstacle being directly behind a grass mound that the robot saw and thought it could run over. Modifications to the training algorithm could avoid many of these instances by having the simulated vehicle prefer not to run over vegetation when there was any other clear path (train the network to avoid unseen areas). For this test, we add two new metrics beyond the deviation from the IODA's steering direction: 1) Average distance from the nearest obstacle, & 2) The number of collisions with obstacles.

7.6.2.9.1 Data gathering and training

Training progressed in the same manner as described in the first Simulated Slag Heap Conflicting Obstacle Experiment. We used the IODA to generate a training set for the Navlab II Monolithic neural network, and then one for each FeNN. ANGEL was used to find the correct number of hidden units for optimal results. The monolithic network required 6 hidden units and 2000 epochs to

reach peak performance on its test set. The color FeNN required 600 epochs and 6 hidden units, and the range FeNN required 250 epochs and 9 hidden units. The Task Net required 2300 epochs and 10 hidden units. Thus, the monolithic network took 22,320,000 weight adjustments, and the MAMMOTH network took 3,900,600 weight adjustments (the color FeNN took 3,279,600 and the Task Net took 621,000; the range FeNN took only 2,328,750 weight adjustments which were done in parallel with the color FeNN's weight adjustments).

7.6.2.9.2 *Results*

The table below shows the results for each of our metrics for the monolithic neural network, the MAMMOTH neural network, and for the Improved Omniscient Driving Algorithm (which provided the training signal). The MAMMOTH network had a 10.3% mean deviation from the IODA over 1 million cycles. The monolithic network had a 10.9% mean deviation from the IODA over 1 million cycles.

Although the difference in performance is small, it is statistically significant. Using standard statistical techniques [Anderson96], we compute the "z value" for comparing two independent sample means for the purposes of detecting the differences between the two populations. The sample variance for the MAMMOTH network was 121.91, and the sample variance for the monolithic network was 166.15. The z-value is defined as the difference of the means over the square root of the sum of the quantity of the variances over the sample sizes:

$$z = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

For our measurements, z = 35.35. For us to be confident at the 1% level that the two means, 10.3 & 10.9 represent truly different populations, z only has to be above 2.576, which it is. Note, however, that although the difference is statistically significant, it is still small.

	Monolithic	**MAMMOTH**	**IODA**
Deviation	10.9%	10.3%	n/a
Avg minimum distance	8.9 meters	8.9 meters	9.1 meters
# of collisions	1066/1,000,000	1232/1,000,000	375/1,000,000

As the chart shows, the MAMMOTH network performed as well as the monolithic network over the 1 million test cycles (which is a 1000km simulated autonomous run). Each came close to or hit roughly one obstacle per kilometer (slightly worse for MAMMOTH), which can be expected given

that we have densely scattered obstacles that can be frequently hidden behind mounds of grass (and also given that even the IODA sometimes hit obstacles). The MAMMOTH network deviated slightly less from the steering directions the IODA would have chosen. The MAMMOTH network simply attained this goal with a much shorter training time.

7.7 How much time do we save?

Throughout this chapter, we have listed the benefit of MAMMOTH neural networks over the similarly performing monolithic neural networks in a comparison of weight adjustments required in order to train a network to optimal performance on a test set. But we also know that we have to spend extra time generating training data for the MAMMOTH FeNNs.

For all of the experiments for which we have computed the comparison, let's consider the following: in training a network for a task, we actually train a number of similar networks with a related number of hidden units in order to find the best architecture for a particular set of data (see Figure 4 on page 16 for a sample output from ANGEL showing different errors for networks with different numbers of hidden units trained on the same data). Even then, we may train several networks of each architecture, but with different random initial weights. Furthermore, the search space for network design and training is much larger than even that would imply. We could try networks with different numbers of hidden layers, different patterns of connectivity, different learning rates, different momentum variables, a variety of weight smoothings (adjusting a weight based not just on the relationship between its source and termination nodes, but also on the weights near it in the same layer), and other variables. The whole search space could be very large indeed.

Let's say, just for illustration, that we only explore 5 different network architectures, and we train five networks with each architecture. Therefore, let us assume we have to train 25 networks whose time to best training roughly averages to that of the best networks we found. Also, we can assume using a MAMMOTH network *adds 2 hours to the data gathering time* for our cross country navigation tasks. Finally, we can use the figure of 5 million weight updates per hour for a very fast workstation. If we have two workstations for training, we have a throughput of 10 million weight adjustments per hour. Please refer to the chart in Figure 46 on page 113 for results.

How much time do we save?

Experiment	Monolithic Weight Adjustments	Modular Weight Adjustments	Time to Train 25 Monolithic Networks	Time to train 25 Modular Networks	Time Savings With Modular Networks
Road-obstacles	> 6.5 million	< 4.5 million	> 16.25 hours	< 11.25 hours	18% (> 3 hours)
Real World Additive Obstacle	> 10.5 million	< 4 million	> 26.25 hours	< 10 hours	54% (> 14.25 hours)
1st Simulated Conflicting Obstacle	> 20 million	< 2 million	> 50 hours	< 5 hours	90% (> 45 hours)
2nd Simulated Conflicting Obstacle	> 22 million	< 4 million	> 55 hours	< 10 hours	82% (> 45 hours)

Figure 46 Time savings with MAMMOTH

The computational and time savings are quite significant when our MAMMOTH Navlab II networks are used.

8 Conclusions & Contributions

8.1 Interpreting the Results

When we began this work, we had two goals: develop solutions for two robotics tasks, and explore a novel modular neural network architecture. In this document we have shown working vision systems for ANDI, a complete navigation system for TIP, and a flexible and trainable multi-sensor navigation system for the Navlab II Unmanned Ground Vehicle. Furthermore, we have identified a task domain in which our novel neural network architecture provides dramatic training time improvements measured in the number of weight adjustments and forward passes through the training set needed to achieve optimal performance for a given network.

8.1.1 Understanding MAMMOTH

The key question to ask is, "How can this work be useful to other researchers?" That question can be answered in part by a simple list of areas to which this work makes some contributions (see "Contributions" on page 118), but another large part of the answer can be found in examining how MAMMOTH succeeded with the Navlab II experiments as opposed to the ANDI experiment.

We believe that the answer comes back to the concept of *functional decomposition*. In particular, we must emphasis the notion of *decomposition*. Simply put, the FeNN in the Edge MAMMOTH Retinal IRA network was used, not as part of a strict decomposition of the mapping from input to output space, but as a supplement to the equally large "glue" section which was free to learn oppor-

tunistically. We only added the need for *additional processing* to the system by adding a FeNN. Although there is some significant evidence (for example, in [Caruana96] and [Waibel89]) that the presence of hidden units influenced by related tasks can help in the ultimate task performance, our ANDI experiments were inconclusive in that regards (due in part to a paucity of good data). The fact that the "glue" portion of the MAMMOTH IRA network was as large as the FeNN meant that when we learned the high level mapping in the network, we were still adjusting weights in a large low level mapping. We could not see any training time reduction in this case where we did not decouple learning the high level and low level mappings.

On the other hand, consider the Navlab II MAMMOTH network architecture. In those networks, we deprived the Task Net(s) of any "glue" or direct access to the raw inputs, and we still managed to get performance that was consistently as good as or better than what we could achieve with monolithic neural networks on the same training data. The simple fact that we could achieve comparable performance meant that we won overall, because we could parallelize the training process and, more importantly, we could separate learning the high level mapping from learning the low level mapping (functional decomposition). For our experiments, the higher level mappings required more training epochs, and not having to perform forward passes through or adjust the weights of the lower levels of the network while training the higher levels resulted in a large time savings. We saw training time reductions of up to 90% measured in terms of the total number of sequential weight adjustments required.

Furthermore, we saw some evidence, especially in both of the simulated Conflicting Obstacle experiments, that breaking down the mapping into simpler parts made the learning of low-level features occur more rapidly (in the two experiments, the FeNNs trained in between 250 and 600 epochs, while the monolithic neural network took 2000 or more epochs to develop its high and low level features). In the second of those two experiments, the Task Net took a large number of epochs to train, but since it was learning from pre-learned FeNN representations, the large number of epochs passed quickly (there were only several hundred connections to train in the Task Net). This lends credence to the idea that separating the training of low-level and high level representations can be beneficial for the training of both.

Therefore, it appears that the key benefits we could derive from MAMMOTH came about when we had knowledge of the task which let us truly break the problem down into simpler components which could be combined to *fully* recreate the more complex mapping. We knew that we could generate training signals for features to be found in the different sensors' images, and we knew that the combination of those features were enough to solve the larger problem.

Perhaps the most important question is, "When does this training time reduction make the biggest difference?" which is similar to "To what tasks is MAMMOTH most applicable?" For problems in which a functional decomposition can be chosen by an expert, MAMMOTH will result in some

time savings when a large space of network parameters such the number of hidden units, the connectivity patterns, or the learning parameters needs to be explored.

The time savings becomes much more significant in problems in which a new training set is needed with some frequency. For example, if we needed a network for navigating around unchanging obstacles using laser range data, we may not need to explore a MAMMOTH solution, since the images generated by digitizing range images of predictable objects would not change often, and the network would not need to be retrained with any frequency. We could train a monolithic network once, pay the one-shot price of a longer training time and not have to retrain.

However, in XC navigation we need to use color video data to see vegetation. Vegetation frequently changes color (and even size and shape). We would need to retrain a network for navigation with some regularity in order to keep up with the current conditions. With a monolithic neural network, we would be paying the training time price repeatedly. In such a case, using MAMMOTH has a great benefit measured in time saved. Furthermore, we don't have to retrain the parts of the network (FeNNs) which correspond to features which have not changed. Thus, we could just retrain a color video FeNN and the Task Net and leave a range FeNN as it was from the original training. The reuse of neural code could be a valuable time saver (note that we did reuse the real world range FeNN from the additive obstacle test for the real world conflicting obstacle test).

Additionally, in one sense, a reduced training time would be equivalent to better performance (a lower error rate) in a case in which the training time for a large, complex network could be prohibitively long. Although in the scope of our research, we have not encountered a network of that level of complexity, it is conceivable that for larger tasks with larger input and output spaces and larger training sets, a monolithic network would be unable to learn the function to its best performance in an acceptable amount of time for a given task. We showed in "Task Performance" on page 108 that a monolithic network that was not fully trained exhibited a markedly worse average deviation from the steering direction selected by the IODA (almost 37% greater error). Thus, for a network that could not be trained to its full potential due to time constraints, performance suffers. A paradigm such as MAMMOTH which reduces training times could result in performance advantages.

Sensing tasks, and tasks which use multiple sensors seems to be well suited to this sort of functional decomposition. But, we believe that the MAMMOTH modular neural network paradigm is not limited in potential to just the tasks we presented and those that are very similar, but that MAMMOTH can have broad applicability to learning any function for which a functional decomposition is discernible.

In summary, MAMMOTH is beneficial when we can decouple the learning of the low and high level mappings (functional decomposition), and MAMMOTH is most useful when the network needs to be retrained frequently. MAMMOTH can significantly reduce training times.

8.2 Contributions

This final section of the dissertation defines what we've presented and shows what our contributions are. This thesis is a presentation of and exploration of a novel modular neural network architecture and training paradigm which has been shown to have significant training time benefits for a number of tasks for our Navlab II vehicle. Additionally, this thesis is a presentation of solutions to two new robotics sensing and control tasks: the alignment of a robot for inspecting rivets, and cross country navigation with an Unmanned Ground Vehicle in the presence of vegetation.

This thesis adds to the research and discourse in a number of different areas. Some of its main contributions are listed below.

8.2.1 Modular Neural Networks

These initial experiments with the novel MAMMOTH modular neural network reinforce or introduce several ideas related to the field of modular neural network research, and add weight to the notion that modularity is beneficial for large tasks.

- **Informed functional decomposition.** With MAMMOTH we use modular neural networks that use the modules to solve part of the whole task, as opposed to traditional modular techniques in which each module performs task decomposition (solves the whole task for some set of inputs). This decoupling of the learning of low and high level mappings results in a system which has been shown for certain tasks to learn a complicated mapping in a greatly reduced number of weight adjustments and forward passes through the training set.
- **Radically different low-level modules.** MAMMOTH is an example of a successful modular neural network paradigm which allows the modules to be architecturally different, so we don't have to try to force our data to be homogeneous. Thus, we are among the first to show a modular paradigm which is proven to be well suited for sensor fusion problems in which the sensor data is not homogeneous.
- **Integration of a priori knowledge.** Although we hoped to show that our knowledge of edges as useful rivet subfeatures would conclusively help in the rivet finding task, we believe that the general idea of functional decomposition involves injecting task knowledge (or a priori knowledge) into the neural network learning. For MAMMOTH this means that the expert uses their knowledge of a task to perform the functional decomposition which decouples learning the high and low level mappings. This results in a training time reduction.

8.2.2 *Sensing and Sensor Fusion*

This thesis adds to the discourse in three areas relating to sensor fusion.

- **Efficient sensor fusion for real-time robotics.** By performing our sensor fusion on pre-learned features, we avoid the need to fuse all of the raw data. This can lead to a reduced processing time for higher level mappings from the fused data.
- **Implicit calibration.** In this work, the calibration issues are implicit to the control functions. This, too, helps us avoid unnecessary work, as we fuse only what we use.
- **Virtual sensors.** The MAMMOTH paradigm lets us take advantage of our world knowledge to make tasks easier to solve and mappings easier to learn. Our Task Nets do not have to map from real-world raw sensor data, but from virtual sensor data that represents features we know to be useful. If we can establish good virtual sensors, we have essentially decoupled learning the high and low level mappings.

8.2.3 *Systems Implemented & Tasks Addressed*

One of our largest contributions is to have implemented several real world systems.

- **Rivet Finding.** In our rivet finding tasks we demonstrated the usefulness of a monolithic neural network technique for all of our data. Our system's strength is that it is trainable and can be re-implemented in different environments quickly and can take advantage of whatever features are useful for finding the principle targets (rivets in this case). On TIP we built a complete alignment system around our successful rivet hole finder.
- **Road Navigation.** Our accomplishment in this realm was to take some initial steps from road following to road navigation. By adding range data to the color video data and applying a MAMMOTH neural network, we were able to develop a successful road navigator that could follow a road and avoid immobile obstacles in its path. Also, the real world additive obstacle experiment involved many of the same sorts of problems as road navigation entails (avoiding grass while avoiding obstacles). This is the first fully trainable extension of ALVINN that attempts to handle obstacles in the road in addition to road navigation.
- **Cross Country Navigation.** What we accomplished for cross country navigation was to build the first system which successfully fused color video and laser range data for reactive navigation in the presence of vegetation. We needed color video data to detect the vegetation, and we needed range data to detect physical obstacles. By using FeNNs to generate low-level features, we were able to find features in each sensor modality relevant to navigation tasks. Our Task Nets were then used to integrate this data successfully enough to solve the Additive Obstacle task at the

Slag Heap and the Conflicting Obstacle Task in our simulator. This is the first fully trainable system for XC navigation using both color and range data, and one of its main advantages is that multiple sensor modalities can be integrated without explicit calibration or registration.

8.2.4 *Robotics*

The nature of the MAMMOTH system is that its inputs and outputs are not inherently bound to a single task. We have provided examples of MAMMOTH solving real world and simulated tasks, and we have also shown some examples of how to imbed a modular neural network in a real world robot. For example, MAMMOTH, as implemented with INNNAV, takes advantage of multiple computers to process data from multiple sensors in a real time system. We look forward to applying our principles to new robotics tasks, as well as to refining the performance of our systems on our current robotics tasks (see the chapter, "Future Directions" starting on page 121). We believe that a major contribution of this system to robotics in general is that it has been demonstrated to be useful for controlling a robot by fusing data from multiple sensor modalities without needing explicit calibration and registration.

A Future Directions

This thesis stands as a complete unit, but both the problems addressed and the techniques used to address them stand as works in progress. Several obvious next steps are listed below. This list may not be exhaustive, but it highlights important areas researchers might want to investigate to continue this work. Some of the next steps are fairly straightforward, and some may represent a very long commitment.

1.1 MAMMOTH Road Following

One of the areas that we became more interested in as this work developed was he MAMMOTH architecture as applied to road navigation tasks. The road navigation was originally a side issue for us, but the environment on-road seems well suited for developing robust FeNNs for following the road (as ALVINN has shown) and for avoiding obstacles [Pomerleau91]. The tasks involved here are complex enough that modelling them can be difficult and constrained enough that generating training signals that span the whole space of possible inputs is achievable. There are a number of common features in the road environment which could be the subject of FeNNs (guard rails, lines, cars, trucks, people), and this makes Road Navigation an area which could help define the nature of FeNNs within a single sensor modality.

1.2 Cross Country Navigation

For XC Navigation, there are a number of issues. Although we identified and solved parts of two key problems in cross country navigation (the additive obstacle task and conflicting obstacle task), there are a large number of steps to take between our system and completely solving the cross country navigation task. Cross country navigation in the presence of vegetation is simply a much harder problem than road navigation in that there are a larger number of variables in the environment and a less clear notion of where to drive to from a given position.

- **Richer training sets.** In the simulator, it was easy to generate large training sets which spanned the space of likely inputs and outputs. In the real world, much more is variable. The features and obstacles change size and shape more often than in the simulator. The changes in lighting over the course of just hours are significant. Vegetation comes in many colors. Generating a large training set is extremely tedious for cross country navigation tasks and we cannot rely on tricks such as generating shifted images from real images (see Figure 22, "Why We Can't Shift Images To Generate Larger Training Sets (view from top)," on page 72). The solution to generating the best training sets is open. It may involve using a model based system to train particular FeNNs off-line. It could involve merely a strict training regimen.
- **Developing FeNNs.** As with the Road Navigation tasks, there are a number of more specific FeNNs which could be developed. Using many FeNNs that found certain classes of obstacles (perhaps by size or shape) could reduce the burden of generating huge training sets for just a few FeNNs. Different kinds of plants, rocks, and man-made artifacts could all be the subject of FeNNs
- **Artificially generated FeNNs.** Another way to generate very good FeNNs might be to use FeNNs trained on a very realistic simulator. We could train a FeNN to mimic the Improved Omniscient Driving Algorithm in the real world by training it with the IODA in the simulator. This sort of technique could be interesting when we have a good model for navigation that requires omniscience not available in the real world.
- **Engineering the data.** One very important issue is engineering or conditioning the raw data. For example, we could eliminate some of the concerns of varying lighting conditions by normalizing the color video images.
- **Integration into higher level control.** Ultimately, for the system to be useful it has to be more than just a wander-mode system. The vehicle has to be able to go to desired locations. The output signal we have used is less than ideal for this sort of integration in that our output usually indicates only the single best steering direction. An output signal that ranks various steering directions could be used for integration. Or a network could be trained with the desired steering direction as an additional input (though this, of course, would make the needed training set even larger and more difficult to develop).

1.3 *Evolving MAMMOTH itself*

MAMMOTH, as presented in this thesis, is an initial exploration of modular networks that perform functional decomposition. In fact, MAMMOTH can be thought of simply as monolithic networks for pre-processing (the FeNNs) and monolithic networks for higher level processing (the Task Nets). There are an enormous number of variants that could be made to the MAMMOTH architecture and the general application of MAMMOTH to various tasks.

- **Not frozen FeNNs.** In this work, we choose to use FeNNs whose weights were frozen before the Task Net was trained. This decision was done so that we could be sure that our task level inputs were the features we wanted them to be. However, it may be very useful to allow the FeNNs' weights to change with the Task Net's weights. Thoroughly examining the effect of this would be a very interesting study. This idea has worked well in the past [Waibel89], and could be successful with MAMMOTH.
- **Input reconstructing FeNNs.** FeNNs modelled after Gluck's Hippocampal networks [Gluck93] could be used to develop novel features common to multiple sensor modalities (or virtual sensors).
- **Asynchronous multi-modality integration.** A modification that could benefit many multi-sensor tasks would be allowing the data from the various sensors to propagate through the neural networks at whatever rate it comes in. The benefit would be faster system cycling. The network would have to have some input reflecting the state of the system with regards to each sensor's data acquisition, so that it could accurately extrapolate from the latest data.
- **State FeNNs.** One of the limitations of the MAMMOTH networks found in this document is that they are exclusively reactive. That means that features which are not currently visible to the sensors are not part of the training signal. For obstacle avoidance tasks on the Navlab II this was a problem because the sensors would frequently get past an obstacle before the vehicle cleared it. Inputs to the MAMMOTH network which include state data (such as features from past cycles of the FeNNs, or past steering directions) could be used to increase the reliability of the system.

B Appendix: ANGEL

Although the techniques described in the rest of the document form the nucleus of this thesis, this work would not be a complete robotics thesis if we did not include a description of one of the most useful tools we had: ANGEL, the Automatic Network Generator & EvaLuator.

2.1 ANGEL Neural Network Generator (Automatic Network Generator & EvaLuator)

We have a system that allows us to design and implement a single neural network very quickly (derived from the same system used in [Pomerleau91], but we frequently need to be able to test large numbers of networks on the same data to find the best network architecture for a given task. Towards this end, we designed and implemented ANGEL, an extremely flexible multi-computer neural network generator, trainer, and evaluator.

The advantage of ANGEL over simply starting multiple copies of the single neural network code is that ANGEL let's us specify a general neural network architecture and it will automatically start training different variants of that architecture on different machines. In our lab there are approximately fifty Sun workstations. ANGEL builds and maintains a list of workstations that nobody else is using, and whenever a new network needs to be trained, ANGEL finds the fastest available work-

station to train it on. Moreover, if a fast workstation becomes free (the user had logged out), ANGEL can halt a network on a slower machine and move its training to the faster one. ANGEL also monitors each machine on which it is training a neural network, and if someone logs in to use that machine, ANGEL will gracefully log out and start the net that was there on the next fastest available machine. Often, ANGEL will be monitoring and training twenty networks all at once. The final good point of ANGEL is that it monitors and records the performance of each network on an independent test set at all stages of training.

To use ANGEL, the user provides a basic neural network descriptor file, training data & test data, a set of evaluation routines, and routines to choose a new network to test. For our purposes so far, ANGEL has only done exhaustive searches on a small number of hidden units for a given architecture (that is, we supply a training set, and ANGEL trains networks with every number of hidden units from, for example, one to twenty). The descriptor file consists of pointers to the training and test sets, as well as the dimensions of the inputs and outputs, connectivity patterns, and ranges of hidden units to try (as well as the number of different networks with the same number of hidden units but different initial random weights to try for each valid number of hidden units).

ANGEL consists of five primary modules: the NetMaker, the Executor, the Trainer, the Monitor, and the User Interface. Each runs as a separate process, and they communicate over the ethernet with each other and with the networks-in-training.

2.1.1 NetMaker

The NetMaker's job is to come up with the next neural network architecture to test, to generate all of the appropriate files and data, and to alert the Executor that the next network is ready. The NetMaker can receive a command to make a new net only from the Monitor which (among other things) keeps track of the maximum number of networks we let ourselves have training simultaneously.

The NetMaker has two parts, the Chooser & the Maker. The Chooser is task specific, and its job is to decide on the next network to implement based on what has occurred so far in ANGEL. Currently, the Chooser simply keeps track of the last number of hidden units and increments it each time a new network is requested; then the Chooser generates a description of the new network and sends it to the Maker. The Maker takes the basic description from the Chooser and turns that description into all of the relevant files. One goal is to conserve storage space by using symbolic links to all of the large data files such as the training and test sets. The Maker generates the files and communicates with the Executor to tell it to start a new training process.

2.1.2 *Executor*

Once a network's files have been generated, a message gets sent by the NetMaker to the Executor. The Executor maintains two queues. One is a queue of networks to start running and the other is a queue of available machines (sorted in order of speed).

The Executor starts with a list of all CPUs on the network on which the user will allow nets to be trained. The Executor checks each machine to see if that machine is in use. If the machine is in use, it rechecks it later (at a randomly generated time interval) to keep the list up to date. When the Executor receives a command to start a new network. it chooses the first machine on its available-machine queue, double checks to make sure nobody has logged in yet, and then forks off a process which remotely logs into the selected machine and starts the new network training.

2.1.3 *Trainer*

There is a Trainer module for each network being trained, each running on its own machine. The first part of a Trainer is a system based on the single network training code [Pomerleau91]. The Trainer performs training epochs on the training data while checking for interrupting signals from the Monitor.

The second part of the Trainer is a fitness evaluator which runs whatever routines are provided by the user on the partially trained networks and sends updates to the Monitor. The Trainer tells the Monitor what the current "fitness" of the network is (for display and recording), as well as how far along the training the network is.

2.1.4 *Monitor*

The Monitor maintains a list of all networks which have been successfully started on remote CPUs. It keeps track of data such as fitness metrics, number of epochs, a short net description, and time since last update from each network's Trainer. The Monitor's current implementation runs each network until a maximum number of epochs has been passed or another user has logged into a network's remote machine. When another user has logged in, the Monitor tells the Trainer on that machine to quit, and informs the Executor that network needs to be restarted somewhere else (saving the current state, of course). The Executor is actually blind as to whether the NetMaker or Monitor sent a "start this net" command. Future implementations may allow the Monitor to kill off networks whose current fitness is low, or to stop training if the fitness is not improving any more.

2.1.5 *User Interface*

The final module is a spartan user interface process which allows the user to start the system, monitor each module's status, and manually kill off and restart particular networks.

2.1.6 *The Future of ANGEL*

ANGEL could easily be converted to perform a more complicated search. By modifying just the Chooser part of the NetMaker, the fitness evaluator in the Trainer, and by adding some capabilities to the Monitor, ANGEL could perform genetic searches not just on the network architecture, but also on a whole range of parameters. Learning parameters, connectivity, architecture, resolution of input and output data, automatic FeNN generation, and FeNN-mutation could all be explored.

C Glossary & Acronyms

ALV	Autonomous Land Vehicle
ALVINN	Autonomous Land Vehicle In a Neural Network
ANDI	Autonomous NonDestructive Inspector of aging aircraft
ANN	Artificial Neural Network
CMRI	Carnegie Mellon Research Institute
FDN	Feature Driven Neural Methodology
FRC	Field Robotics Center
HMMWV	High Mobility Multi-Wheeled Vehicle; a military 4-wheel drive, all-terrain, ambulance. Sometimes called a "Hummer." HMMWV is pronounced, "Humvee."
IAN	It's A Name

IODA	Improved Omniscient Driving Algorithm
IRA	Instant Rivet Announcer
Malsim	Modified AL's SIMulator; our 3D simulator for testing the HMMWV
MAMMOTH	Modular Architecture Multi-MOdality Theory
NavLab II	Navigation Laboratory II, aka the HMMWV
ODA	Omniscient Driving Algorithm
RI	Robotics Institute
UGV	Unmanned Ground Vehicle
Virtual Sensor	An abstraction of a sensor that performs some transformation on the raw data of a sense space or which extracts higher level features from raw sense data

D References

[Anderson96] T.W. Anderson and J.D. Finn, *The New Statistical Analysis of Data*, Springer, New York, 1996.

[Ballard82] D. Ballard and C. Brown, *Computer Vision*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[Boyan91] J. Boyan, "Modular Neural Networks for Learning Context-Dependent Game Strategies," Cambridge University Masters Thesis, 1992.

[Brooks89] R. Brooks and A. Flynn, "Fast, Cheap, and Out of Control: A Robot Invasion of the Solar System," *J. British Interplanetary Society*, 42 (10): 478-485, 1989.

[Brumitt92] B. Brumitt, R. Coulter, A. Stentz. "Dynamic Trajectory Planning for a Cross-Country Navigator," *Proc. of the SPIE Conference on Mobile Robots*, 1992.

[Canny83] J. Canny, "Finding Edges and Lines in Images," MIT AI Laboratory Technical Report 720, June 1983.

[Caruana95] R. Caruana, "Learning Many Related Tasks at the Same Time With Backpropagation," *Advances in Neural Information Processing Systems 7*, pp. 656-664, 1995.

[Caruana96] R. Caruana, "Algorithms and Applications for Multitask Learning," L. Saitta (ed.), *Machine LEarning: Proceedings of the Thirteenth International Conference*, San Francisco, CA, 1996.

[Caudill90] M. Caudill and C. Butler, *Naturally Intelligent Systems*, The MIT Press, Cambridge, MA, 1990.

[Coulter91] R. Coulter, "Vision for ROSA," Carnegie Mellon University Technical Report, CMU-RI-TR-91-24, 1991.

[Courtney95] J. Courtney and A. Jain, "Mobile Robot Localization via Classification of Multisensor Maps," *Proceedings of the IEEE Intelligent Vehicles '95 Symposium*, Detroit, USA, pp 1672-1678, 1995.

[Cybenko89] G. Cybenko, "Designing Neural Networks," Center for Supercomputing Research and Development Report 934, Urbana-Champaign, Illinois, 1989.

[Daily88] M. Daily, et al., "Autonomous Cross Country Navigation with the ALV," *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*: 718-726, 1988.

[Davis93a] I. L. Davis and M. W. Siegel, "Automated Nondestructive Inspector of Aging Aircraft," *International Symposium on Measurement Technology and Intelligent Instruments*, Huazhong University of Science and Technology, Wuhan, Hubei Province, People's Republic of China, October 1993.

[Davis93b] I. L. Davis and M. W. Siegel, "Visual Guidance Algorithms for the Automated Nondestructive Inspector of Aging Aircraft," *SPIE Conference on Nondestructive Inspection*, San Diego, July 1993.

[Duda73] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*, Wiley and Sons, 1973.

[Durrant-Whyte88] H. Durrant-Whyte, *Integration, Coordination and Control of Multi-Sensor Robot Systems*, Kluwer Academic Publishers, Boston, 1988.

[Fahlman90] S. E. Fahlman and C. Lebiere, "The Cascade Correlation Learning Architecture," Carnegie Mellon University Technical Report, CMU-CS-90-100, 1990.

[Fahlman91] S. E. Fahlman, "The Recurrent Cascade Correlation Architecture," Carnegie Mellon University Technical Report, CMU-CS-91-100, 1990.

[Gluck93] M. Gluck and C. Myers, "Hippocampal Mediation of Stimulus Representation: a Computational Theory," To appear in *Hippocampus* (in press 1993), 1993.

[Hager90] G. Hager, *Task-Directed Sensor Fusion and Planning*, Kluwer Academic Publishers, Boston, 1990.

[Hampshire89] J. B. Hampshire and A. H. Waibel, "Connectionist Architectures for Multi-Speaker Phoneme Recognition," Carnegie Mellon University Technical Report, CMU-CS-89-167, 1989.

[Hampshir89b] J. B. Hampshire and A. H. Waibel, "The Meta-Pi Network: Building Distributed Knowledge Representations for Robust Multi-Source Pattern Recognition," IEEE-PAMI, 1989.

[Haussler89] D. Haussler, "Generalizing the PAC model for neural net and other learning applications," Technical Report, University of California, Santa Cruz, UCSC-CRL-89-30, 1989.

[Jacobs90] R. A. Jacobs, M. L. Jordan, and A. G. Barto, "Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks," COINS Technical Report 90-27, 1990.

[Jacobs91] R. A. Jacobs, M. L. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive Mixtures of Local Experts," *Neural Computation*, 3:79-87, 1991.

[Jochem93] T. Jochem, D. Pomerleau, C. Thorpe, "MANIAC: A Next Generation Neurally Based Autonomous Road Follower," *Proc of International Conference on Intelligent Autonomous Systems* (IAS-3), Feb 15-18, 1993.

[Jochem93] T. Jochem, "Using Virtual Active Vision Techniques to Improve Autonomous Road following Tasks," Carnegie-Mellon University Ph. D. Thesis Proposal, 1993.

[Karnin89] E. Karnin, "A Simple Procedure for Pruning Back-Propagation Trained Neural Networks," Research Report IBMC 14834, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1989.

[Keller87] J. M. Keller, R. M. Crownover, and R. Y. Chen, "Characteristics of Natural Scenes Related to the Fractal Dimension," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, No. 5, September, 1987.

[Kelly92] A. Kelly, A. Stentz, M. Hebert, "Terrain Map Building for Fast Navigation on rugged Outdoor Terrain," *Proc. of the SPIE Conference on Mobile Robots*, 1992.

[Kelly93] A. Kelly, "A Partial Analysis of the High Speed Cross Country Navigation Problem," Carnegie-Mellon University Ph. D. Thesis Proposal, 1993.

[Knight89] K. Knight, "A Gentle Introduction to Subsymbolic Computation: Connectionism for the A.I. Researcher," Carnegie Mellon University Technical Report, CMU-CS-89-150, 1989.

[Kweon91] I.S. Kweon, *Modelling Rugged Terrain by Mobile Robots with Multiple Sensors*, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, 1991.

[Langer93] D. Langer, J. K. Rosenblatt, M. Hebert, "A Reactive System For Off-Road Navigation," Carnegie Mellon University Technical Report, CMU-RI-TR-93-, 1993.

[Latombe91] J.-C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, 1991.

[Lim93] S.F. Lim and S.B. Ho, "Dynamic Creation of Hidden Units with Selective Pruning in Backpropagation," National University of Singapore Tech Report D4/93, 1993.

[Marra88] M. Marra, R. T. Dunlay, and D. Mathis, "Terrain Classification Using Texture for the ALV," Proceedings of the SPIE Conference on Mobile Robots, 1992.

[Mars88] Mars Rover/Sample Return (MRSR) Rover Mobility and Surface Rendezvous Studies, Final Report, Martin Marietta, JPL Contract No. 958073, October 1988.

[Matthies92] L. Matthies, "Stereo Vision for Planetary Rovers: Stochastic Modelling to Near Real-Time Implementation," International Journal of Computer Vision, Vol 8, No. 1, pp 71-91, 1992.

[Meeden93] L. Meeden, G. McGraw, and D. Blank, "Emergent Control and Planning in an Autonomous Vehicle," *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, 1993.

[Minsky69] M. Minsky and S. Papert, "Perceptrons; an introduction to computation geometry." MIT Press, Cambridge, MA, 1969.

[Pomerleau95] D. Pomerleau, "RALPH: Rapidly Adapting Lateral Position Handler," *Proceedings of IEEE Intelligent Vehicles '95 Symposium*, Detroit, USA, 1995, pp. 506-511.

[Pomerleau92] D. Pomerleau, *Neural Network Perception for Mobile Robot Guidance*, Ph.D. Dissertation, Carnegie-Mellon University Technical Report CMU-CS-92-115, 1992.

[Pomerleau91] D. Pomerleau, "Neural network-based vision processing for autonomous robot guidance," *Proceedings of SPIE Conference on Aerospace Sensing*, Orlando, FL, 1991.

[Press90] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vatterling, *Numerical Recipes in C*, Cambridge University Press, Cambridge, 1990.

[Rumelhart86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. E. Rumelhart and J. L. McClelland, Ed. MIT Press, 1986.

[Shafer86] S. Shafer, A. Stentz, C. Thorpe, "An Architecture for Sensor Fusion in a Mobile Robot," Carnegie Mellon University Technical Report, CMU-RI-TR-86-9, 1986.

[Smieja92] F. J. Smieja and H. Mühlenbein, "Reflective Modular Neural Network Systems," Technical Report, German National Research Centre for Computer Science, March, 1992.

[Smith86] R. C. Smith and P. Cheeseman, "On the Representation of Spatial Uncertainty," The International Journal of Robotics Research, Vol. 5, No. 4, Cambridge, MA, 1986.

[Stentz93] A. Stentz, "Optimal and Efficient Path Planning for Unknown and Dynamic Environments," Carnegie Mellon University Technical Report, CMU-RI-TR-93-20, 1993.

[Tabakman93] T. Tabakman and I. Exman, "Towards Real-Time Self-Organizing Maps with Parallel and Noisy Inputs." Technical Report 93-13, Institute of Computer Science, The Hebrew University of Jerusalem, 1993.

[Thompson77] A. Thompson, "The Navigation System of the JPL Robot," *Proceedings of the International Joint Conference for Artificial Intelligence*: 749-757, 1977.

[Tsai88] R. Tsai, "Overview of a unified calibration trio for robot eye, eye-to-hand, and hand calibration using 3D machine vision," *Research Report RC*. International Business Machines Corporation, Research Division; RC 14218, 1988.

[Tsai86] R. Tsai, "Review of the two-stage camera calibration technique plus some implementation tips and new techniques for center and scale calibration," *Research Report RC*. International Business Machines Corporation, Research Division; RC 12301, 1986.

[Waibel89] A. Waibel, "Modular Construction of Time-Delay Neural Networks for Speech Recognition," *Neural Computation 1*, pp. 39-46. Cambridge, MA, 1989.

[Wasserman89] P. Wasserman, *Neural Computing: Theory and Practice*, Van Nostrand Reinhold, New York, 1989.

[Wilcox87] B. Wilcox, et al., "A Vision System for a Mars Rover," *SPIE Mobile Robots II*. Cambridge, MA, 1987.

[Wright89] W. A. Wright, "Contextual Road Finding With A Neural Network," Technical Report, Sowerby Research Centre, Advanced Information Processing Department, British Aerospace, 1989.