UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

IMPLICIT ROBOT LOCALIZATION THROUGH PREDICTION

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

JOSHUA J. BEITELSPACHER
Norman, Oklahoma
2006

IMPLICIT ROBOT LOCALIZATION
THROUGH PREDICTION

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Prof. Dean Hougen, Chair

Prof. Amy McGovern

Prof. Andrew Fagg

© Copyright by JOSHUA J. BEITELSPACHER 2006
All Rights Reserved.

Dedication

To my parents, David and Kathy Beitelspacher.

Acknowledgments

I would like to thank my advisor, Dr. Dean Hougen, for providing guidance from the very beginning of my work. I would also like to thank Dr. Amy McGovern and Dr. Andrew Fagg for serving on my thesis committee. The insights they provided greatly improved the quality of this thesis.

The members of the Robotics, Evolution, Adaptation, and Learning Laboratory also deserve thanks for their helpful suggestions. Particularly, Mark Woehrer pointed me in interesting and unexpected directions that greatly influenced the final outcome of my work.

Finally, I would like to thank my parents for making this possible, and Kiley for helping me through the final months of work.

Contents

Acknowledgments **iv**

List Of Figures **vii**

Abstract **ix**

1 Introduction **1**
1.1 Localization as a System Property 1
1.2 Overview 2

2 Neural Networks and Reinforcement Learning **4**
2.1 Neural Networks 4
2.1.1 Feed-Forward Networks 5
2.1.2 Recurrent Networks 9
2.1.3 Self-Organizing Maps 11
2.2 Reinforcement Learning 12
2.2.1 Action-Value Functions 15
2.2.2 Q-Learning 18

3 Mobile Robot Localization **21**
3.1 Localization Strategies 22
3.1.1 Explicit Localization 22
3.1.2 Implicit Localization 25
3.2 Localization Problems 27
3.2.1 Position Tracking 28
3.2.2 Global Localization 28
3.2.3 Simultaneous Localization and Mapping 29
3.2.4 Generalizations to Implicit Localization 30

4 Developing Implicit Localization **32**
4.1 Developmental Robotics 32
4.2 Bootstrapping Learning 36
4.3 Learning Abstraction 37

4.4 Learning Anticipation . . . 41
4.5 Improving Performance . . . 50
4.6 Application to Localization . . . 53

5 Evaluating Implicit Localization 57
5.1 Path Following . . . 57
5.2 Grid World . . . 58
5.2.1 Anticipation . . . 60
5.2.2 Localization along a Path . . . 72
5.2.3 Control from Prediction . . . 79
5.2.4 Reinforcement Learning . . . 82
5.3 Simulated Robot . . . 87
5.3.1 Abstraction . . . 89
5.3.2 Anticipation . . . 90
5.3.3 Localization along a Path . . . 95
5.3.4 Control from Prediction . . . 98
5.3.5 Reinforcement Learning . . . 101

6 Discussion 104
6.1 Achievements . . . 104
6.2 Contributions . . . 107
6.2.1 Developmental Robotics . . . 107
6.2.2 Learned Robot Navigation . . . 108

7 Conclusion 111
7.1 Summary . . . 111
7.2 Future Work . . . 112

Reference List 114

List Of Figures

2.1 A single perceptron 5
2.2 A feed-forward neural network 6
2.3 Propagation algorithm 7
2.4 Backpropagation algorithm 7
2.5 An Elman network 10
2.6 Reinforcement learning 13
2.7 State, actions, and rewards 13
2.8 A grid world task 14
2.9 Grid world task solution 17
2.10 Q-Learning algorithm 19

3.1 Example metric map 23
3.2 Example topological map 24

4.1 Basic developmental system 34
4.2 Proposed abstraction layer 40
4.3 Predictions from a multi-step prediction network 42
4.4 Proposed abstraction and anticipation system 44
4.5 Abstraction training algorithm 44
4.6 Anticipation training algorithm 45
4.7 Alternating sequence 1-step prediction training 47
4.8 Alternating sequence 9-step prediction training 47
4.9 Predictive control algorithm 48
4.10 Offline prediction error computation 51
4.11 Online prediction error computation 52
4.12 Paths through an indoor environment 55

5.1 Grid world environment 59
5.2 Overlapping grid world path 61
5.3 Different grid world path 62
5.4 Grid world training 1-step prediction and 20 hidden nodes 64
5.5 Grid world training 1-step prediction and 40 hidden nodes 65
5.6 Grid world training 1-step prediction and 80 hidden nodes 66
5.7 Grid world training 10-step prediction and 20 hidden nodes 67

5.8 Grid world training 10-step prediction and 40 hidden nodes 68
5.9 Grid world training 10-step prediction and 80 hidden nodes 69
5.10 Grid world offline prediction errors 71
5.11 Grid world online prediction errors 74
5.12 Activation of hidden node 17 on 6 grid world paths 75
5.13 Activation of hidden node 22 on 6 grid world paths 76
5.14 Activation of hidden node 28 on 6 grid world paths 77
5.15 Grid world path following 1-step prediction 80
5.16 Grid world predictive control performance 81
5.17 Grid world Q-Learning performance 86
5.18 Simulated robot environment 87
5.19 Sonar abstraction 90
5.20 Simulated robot 1-step prediction training 91
5.21 Simulated robot 80-step prediction training 92
5.22 Simulated robot 1-step offline prediction errors 93
5.23 Simulated robot 80-step offline prediction errors 94
5.24 Simulated robot 80-step 80-hidden node online prediction errors . . . 95
5.25 Different path for simulated robot 96
5.26 Simulated robot prediction errors on a different path 97
5.27 Similar path for simulated robot 97
5.28 Simulated robot prediction errors on a similar path 98
5.29 Memorizing motor control along a path 99
5.30 Robot control from prediction 100
5.31 Simulated robot reinforcement learning 102
5.32 Q-Learning path following 103

Abstract

Robot localization is traditionally achieved using explicit map-based representations. However, this approach tends to be task and environment dependent. Different localization algorithms are typically needed for robots that operate in different environments, use different sensor modalities, or have different degrees of freedom. The design of these algorithms is often similar, but there are enough differences to make the transitions difficult and time consuming. To shift the burden from robot designers to the robots themselves, we propose to remove the dependence on known representations and to let robots develop implicit internal representations. Implicit localization is developed through the processes of abstraction and anticipation. Evaluation is performed by measuring performance at the given task. This formulation allows localization to be learned in a general manner, as needed by the task at hand. This approach simplifies robot design and allows for the creation of more flexible robots.

Chapter 1

Introduction

Robot localization is a foundational problem in mobile robotics. All but the simplest mobile robots need to perform some form of localization. The best known localization techniques use advanced algorithms that probabilistically integrate sensor information into a model of the world. While these techniques work very well, they also tend to be complicated. We propose to construct a localization system using no domain specific knowledge. Such a system should work in a wide variety of environments with only simple changes.

1.1 Localization as a System Property

Most research into robot localization is focused on localization as a goal in and of itself. By focusing only on localization it is possible to develop algorithms that can be used by robots to carry out a wide variety of complex tasks. Given a working localization system, it is significantly easier to develop a robot that performs useful behaviors.

However, the ability to perform a desired behavior is more important than localization alone. The ability to localize is a necessary property of a system that performs complex navigation tasks, but it is not the primary purpose of the system. Instead of using a carefully engineered algorithm for localization, we propose to design a system capable of building an internal representation of a task. If the task requires localization, then the internal representation must include information necessary for localization.

We will attempt to design such a system using developmental processes and machine learning. Unlike traditional localization, such a system is incapable of reporting its position to a person in an explicit, predefined representation. Instead, it performs localization only as needed to complete the given task, and the current location of the robot is encoded in an unknown internal representation.

1.2 Overview

Chapter 2 provides background information on the field of machine learning. Artificial neural networks and reinforcement learning are described in detail, and are used throughout the remainder of the thesis.

Chapter 3 describes the basic localization problems. A distinction is made between implicit localization and explicit localization, and motivation for implicit localization is provided.

Chapter 4 considers the design of a system for implicit localization. Developmental robotics is used as a starting point, and an entire system for abstraction and anticipation is presented.

Chapter 5 explores the use of implicit localization in a grid world and on a simulated robot. Results are both presented and analyzed.

Chapter 6 continues the analysis from the previous chapter. Starting from what our results have shown, practical applications for implicit localization are presented. Also, the relationship of this work to previous works concerning both developmental robotics and learned robot navigation is discussed.

Chapter 7 summarizes the major achievements of this work. Areas that need further work are also explored, as both short and long term goals are given.

Chapter 2

Neural Networks and Reinforcement Learning

In order to learn robot localization, we will rely on well-established machine learning techniques. Although these techniques provide a solid backbone, combining them appropriately for a particular task is still a challenge. Machine learning is an expansive field, and we will only discuss two of its subfields here: neural networks and reinforcement learning.

2.1 Neural Networks

Artificial neural networks are connected systems of simple components that derive their expressional power from the manner in which they are connected. Although they are loosely based on biological systems, the biological connection is of little importance to this work.

Rosenblatt (1958) developed the first neural network, the perceptron. The perceptron can be defined by a thresholded dot product operation, but it is more instructive to examine it visually as in Figure 2.1. By adjusting the weight, w_i, associated with input, x_i, the value of the output can be influenced. For any linearly separable

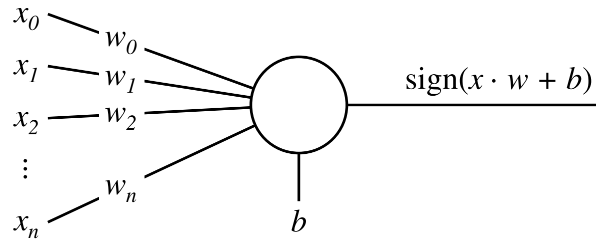


Figure 2.1: The components of a single perceptron unit.

binary function, there is a set of weights, w, to model the function (Minsky and Papert, 1969).

Neural networks have come far from their simple beginnings. Modern network architectures are capable of expressing much more than binary functions. In the remainder of this section we will discuss three distinct types of networks that have emerged.

2.1.1 Feed-Forward Networks

The most common type of neural network is the feed-forward network. Feed-forward networks combine multiple perceptron-like units into an acyclic network. An example network with four input units, three hidden units, and two output units is shown in Figure 2.2. Such a network can be used to approximate a function with four inputs and two outputs.

In order to increase the generality of the network, the perceptron units are replaced with units that output continuous instead of thresholded values. It is still desirable to restrict the values to a known range, so the output of each node can be

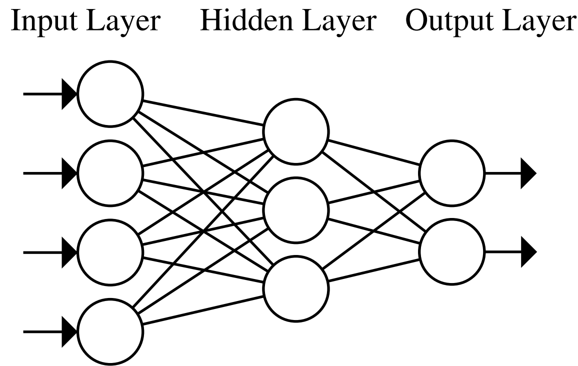


Figure 2.2: A feed-forward neural network. Although the input units are drawn like the other units, they do not do any processing and merely pass data through to the next layer.

passed through an activation function that will limit the range. While any function can be used, sigmoid and tangent functions are particularly popular. Figure 2.3 demonstrates how to compute the output of a network using possibly different activation functions on each layer.

The usefulness of such multi-layer networks was in doubt until a method was developed that could be used to iteratively adjust the weights of the network (Werbos, 1974; LeCun, 1985; Rumelhart et al., 1986). The method, backpropagation, requires an input with a known output in order to modify the weights. Therefore, it is a supervised machine learning algorithm. The full backpropagation algorithm is presented as Figure 2.4. A complete derivation of the algorithm is available in Mitchell (1997), but is not included here. In order to use backpropagation, the activation function for each layer must be differentiable.

Backpropagation is a gradient-descent learning rule. It computes the error at the output nodes by finding the difference between the training data and the network outputs. Further back in the network, error is attributed to the hidden nodes based

```
PROPAGATION
    I = network input
    l = number of layers of weights
    W = array of weight matrices for each layer (W_ijk is the weight
            on layer i for the connection from node k on the previous
            layer to node j on the next layer)
    O = array of output vectors (column vectors) for each layer
    F = array of activation functions for each layer
    O_1 = I
    for i = 1 to l
        O_{i+1} = F_i(W_i O_i)
    return O_{l+1}
```

Figure 2.3: The propagation algorithm for a feed-forward network with fully connected layers and arbitrary activation functions at each layer.

```
BACKPROPAGATION
    I = training input
    T = training output
    α = learning rate
    l = number of layers of weights
    W = array of weight matrices for each layer (W_ijk is the weight
            on layer i for the connection from node k on the previous
            layer to node j on the next layer)
    O = array of output vectors (column vectors) for each layer
    F = array of activation functions for each layer
    E = array of error vectors (column vectors) for each layer
    O_1 = I
    for i = 1 to l
        O_{i+1} = F_i(W_i O_i)
    E_l = F'_l(O_{l+1}) · (T − O_{l+1})
    for i = l to 2
        E_{i−1} = F'_{i−1}(O_i) · (W_i^T E_i)
        W_i = W_i + α E_i O_i^T
    W_1 = W_1 + α E_1 O_1^T
```

Figure 2.4: The backpropagation algorithm for a feed-forward network with fully connected layers and arbitrary activation functions at each layer.

on how strongly they are connected to the each node on the next layer. The network weights are then adjusted in order to reduce the error. The amount to adjust each weight is specified as the learning rate. A learning rate of one attempts to remove all the error in one step, while a learning rate of zero does not adjust the weights. A learning rate close to zero is usually used, and training instances are repeatedly presented to the network. Using a low learning rate causes the weights to be adjusted slowly, and makes the system more likely to converge to a good approximation of the target function. In contrast, a high learning rate encourages drastic adjustments that can make the system unstable.

In certain cases, the backpropagation learning rule will not learn a solution that minimizes the total error. Instead, it learns a solution that minimizes the error locally, but is not a global minimum. A better solution exists, but backpropagation is unable to find it. The use of a *momentum* term is a common solution to this problem. The previous weight adjustment for each weight is stored, and when weights are next adjusted part of the adjustment comes from the previous adjustment. The momentum term is used to scale the previous adjustment. If momentum was one the entire adjustment would be used again, while at 0.5 only half of the previous adjustment is used. Momentum allows backpropagation to skip over local minima without becoming caught and accelerates learning when similar weight changes are repeated. Although useful, momentum can be difficult to apply well. The best value for momentum is task dependent and difficult to estimate (Riedmiller and Braun, 1993).

Feed-forward backpropagation neural networks have shown success at a wide variety of tasks. A backpropagation neural network can be used on almost any task

that can be reduced to function approximation. Rumelhart et al. (1994) provide an overview of the kinds of tasks that are appropriate for backpropagation neural networks.

Backpropagation can be slow to learn. More advanced techniques such as Quickprop (Fahlman, 1988) and RPROP (Riedmiller and Braun, 1993) can be used to increase the learning speed in many problem domains. In this thesis, backpropagation will be used for its theoretical simplicity.

2.1.2 Recurrent Networks

The shortcoming of feed-forward networks is their inability to model sequences. A feed-forward network provides a direct mapping from input to output, but it does not take history into account. Recurrent networks are designed so that the state of the network at previous time steps influences the current network output. This class of neural networks can solve problems that simple feed-forward networks are unable to, but at the cost of additional complexity.

The first proposed method for recurrent networks was Backpropagation Through Time (Rumelhart et al., 1986). This method copies the states of each previous input and hidden layer value. Then, when backpropagation is performed the computed gradient takes into account the current state as well as the history. This method creates significant overhead, and it is only practical if the number of previous states stored is limited.

Elman (1990) proposed that storing just the hidden layer values from the previous timestep would be sufficient for some tasks. This type of network, now known as an Elman network, is the simplest form of a recurrent network. Elman used it

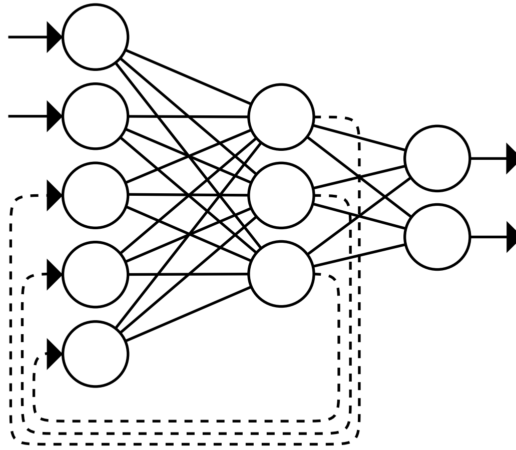


Figure 2.5: An Elman network with two inputs, three hidden units, and two output units. The hidden units' outputs are fed back into the network as inputs.

successfully in work predicting speech patterns. Recently, Elman networks have been used to predict future sensor states on mobile robots (Nolfi and Tani, 1999; Blank et al., 2005). An Elman network is shown in Figure 2.5.

More advanced types of recurrent networks have been proposed. Notably, Real Time Recurrent Networks (Williams and Zipser, 1989) and Long-Short Term Memory (Hochreiter and Schmidhuber, 1997) allow recurrent networks to store more information for longer periods of time than Elman networks or limited Backpropagation Through Time. Recurrent networks are still an area of active research, and only the simplest form of recurrent networks, Elman networks, will be used in the remainder of this thesis.

2.1.3 Self-Organizing Maps

While feed-forward and recurrent networks are both supervised learning systems, self-organizing maps (Kohonen, 1984) are neural network systems that are self-supervised. Much like clustering algorithms, self-organizing maps classify data into a set of groups based on similarity.

Self-organizing maps (SOMs) are defined by a set of neurons arranged in a lattice. The lattice specifies a network topology that can be used to define a neighborhood for each neuron. As in the other types of neural networks, each neuron has a set of weights associated with it. When the network is evaluated for an input, the input vector is compared to the weight vector of each neuron. The neuron with the closest match is the winner. The output of the winning neuron is activated, while all other outputs are cleared.

When training a SOM, the weights of the winning neuron are adjusted so that they more closely match the input. Neurons in the neighborhood of the winning neuron are adjusted toward the input vector to a smaller degree. Over time, the network attempts to learn a mapping such that for any input there will be at least one neuron that closely matches.

While SOMs are not used directly in this thesis, much previous work in the field depends on them (Blank et al., 2005; Provost et al., 2001, 2004, 2006; Madokoro et al., 2003), and any discussion of neural networks would be incomplete without mentioning them.

2.2 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning in which learning occurs through interaction with an environment (Sutton and Barto, 1998). Of all types of machine learning, reinforcement learning is probably the easiest type to explain to people outside the field. From learning to ride a bike to training a dog, people have experienced situations where learning results from interaction. By rewarding good behavior and punishing bad behavior, it is possible to teach an agent to interact intelligently with its environment in order to achieve a goal.

The basic idea of reinforcement learning is quite simple, but the first comprehensive works on the subject did not emerge until the late-1980s (Sutton, 1988), and the most influential books appeared almost a decade later (Sutton and Barto, 1998; Bertsekas and Tsitsiklis, 1996). Although it is now better understood, reinforcement learning is still a relative newcomer to the field of machine learning. The most notable success in RL is the TD-Gammon backgammon player by Tesauro (1995). While there have been many other successes (see Sutton and Barto (1998)), TD-Gammon was able to outperform all other learning techniques and is able to play backgammon better than human grandmasters.

Reinforcement learning algorithms typically model problems as Markov Decision Processes (MDPs). In such a formulation, the problem is defined by states, actions, and rewards. The goal of an agent is to maximize the amount of reward it can receive. Figure 2.6 shows the basic structure of a reinforcement learning system.

An agent interacts with its environment by performing actions in response to state information. After an action is performed the agent receives a reward signal

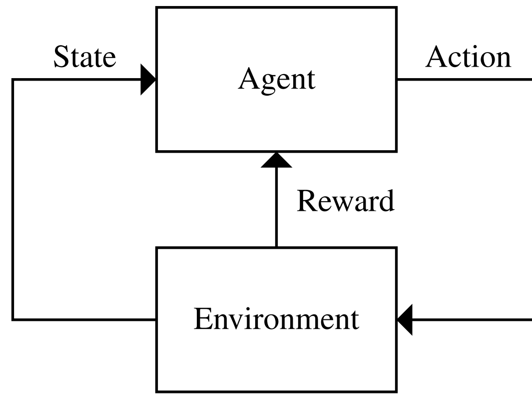


Figure 2.6: A reinforcement learning agent learns through interaction with an environment.

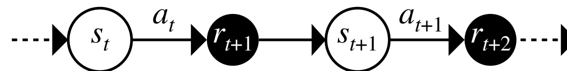


Figure 2.7: A sequence of states, actions, and rewards as observed by an agent.

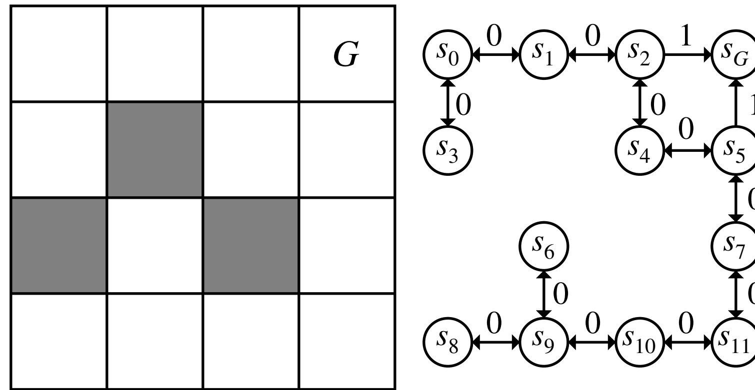


Figure 2.8: A grid world navigation task where the goal is to reach cell G. On the left we have a map of the grid world, and on the right we show the problem as an MDP. Nodes are possible states, edges are possible actions, and edge labels show the reward for each action. Reward is always 0 except for when moving into the goal state, in which case the reward is 1. The task ends when the goal state is reached.

and updated state information from the environment. This process is repeated until the task is finished. Figure 2.7 shows such a sequence of states, actions, and rewards. Over time, the agent attempts to improve its performance by learning which actions lead to rewards. From a state, s_t, the agent wants to choose an action, a_t, that will lead to high future rewards: $r_{t+1}, r_{t+2}, \cdots, r_{t+n}$. This is a difficult problem because the effects of a single action on future rewards is often unclear.

In a simple problem, states, actions, rewards, and the relationships between them may be known in advance. The grid world navigation task in Figure 2.8 is an example of such a task. Figure 2.8 also demonstrates how a task can be represented as an MDP.

2.2.1 Action-Value Functions

When navigating the grid world in Figure 2.8, each action has a known consequence and a known reward. To learn the navigation task, every possible state-action pair is associated with an expected cumulative reward. By convention, we refer to these values as Q-values, and the function that produces them as $Q(s, a)$, where s is a state and a is an action. The action-value function, Q, can be used to guide an agent toward future rewards.

It is also possible to associate reward directly with states, and use a similar function, $V(s)$, to represent the cumulative reward from a particular state. In order to use a value function, V, for control, we need to be able to generate successor states for all possible actions. Using action-value functions instead allows us to easily use these techniques even when the actions have unknown results.

From state s an agent can evaluate all possible actions and choose an action a such that $Q(s, a)$ has the highest possible value. In order to achieve optimal performance, we need an action-value function that exactly predicts the cumulative reward. We will call such a function an optimal action-value function, Q^*. The optimal action-value function defines a *policy* that an agent can follow to achieve maximum reward.

From state s_t an agent takes action a_t, receives reward r_{t+1}, and arrives in state s_{t+1}. As long as we follow the policy given by Q^*, the optimal action-value function gives the sum of all subsequent rewards:

$$Q^*(s_t, a_t) = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_{t+n} \tag{2.1}$$

If the task does not end, some actions may lead to infinite reward. If all possible actions had an infinite expected reward there would be no reason to pick any action over another. We can handle this problem by introducing a discount rate parameter, γ:

$$Q^*(s_t, a_t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} \tag{2.2}$$

The optimal action-value function now gives the discounted cumulative reward. If the discount rate is one, the task is undiscounted. In addition to ensuring that the Q-values never reach infinity, the discount rate also gives preference to reward that is received sooner rather than later. This is often useful because we can be less careful when defining a reward function. For the task in Figure 2.8, moving randomly will eventually reach the goal, so $Q^*(s, a) = 1$ for all s and a if the task is undiscounted. However, by discounting the reward we encourage the agent to reach the goal state quickly.

Because Q^* exactly predicts future rewards, it is possible to write Equation 2.2 as a recursive equation:

$$Q^*(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \tag{2.3}$$

Learning Q^* allows us to have optimal performance at a task. For the task in Figure 2.8, actions that move us into the goal state have a Q^*-value of 1. Q^*-values for actions that are further removed from the goal state can be computed recursively using dynamic programming. The policy derived from Q^* is shown in Figure 2.9.

When state transitions or rewards are unknown, dynamic programming can not be used to find an exact solution to the problem. Even when the system is completely

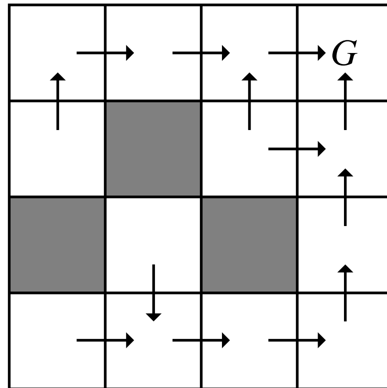


Figure 2.9: A solution to the grid world task.

understood, if there are too many states then dynamic programming can not be used to solve the full problem exactly. In this case, temporal difference (TD) learning can be used in place of dynamic programming to iteratively approach a solution. When using TD learning, differences between successive approximations are used to update the action-value function. At state s_t, action a_t has an expected value of $Q(s_t, a_t)$. If Q is optimal, then $Q(s_t, a_t)$ should exactly equal the reward received, r_{t+1}, plus the discounted future reward, $\gamma Q(s_{t+1}, a_{t+1})$. The TD error is the amount the two values differ. Algorithms that attempt to minimize the TD error are called temporal difference learning algorithms.

Samuel (1959) was the first to use the basic idea of TD learning, and was able to teach an agent to play checkers at a reasonable level. Tesauro (1995) also used a TD learning algorithm for his backgammon agent. These TD algorithms approximated a value function instead of an action-value function because state transitions were known in advance. The remainder of this section gives two specific TD learning algorithms that learn an action-value function.

2.2.2 Q-Learning

The Q-Learning algorithm (Watkins, 1989) maintains an approximate action-value function, and derives its policy directly from the approximated function. If the algorithm always took the optimal action according to the current approximation, it would risk becoming caught in cycles in which the best actions could be overlooked if other actions seemed better. In order to avoid this, it is sometimes necessary to explore the solution space by trying actions which seem less than optimal. This is especially important in non-static environments. Q-Learning is able to do this while still learning an optimal policy.

While there are multiple ways to encourage exploration, a simple and common option is to define a parameter, ε, that gives a ratio defining how often to take random actions. Such a learner is said to follow an ε-greedy policy. It usually takes the greedy action with respect to the current policy, but it randomly takes exploratory actions.

Q-Learning is a TD learning algorithm, so it adjusts its action-value function based on the TD error at each timestep. When learning starts, the Q-function is initialized arbitrarily, and over time it is adjusted to more closely reflect reality. Starting from state s_t, an action a_t is chosen. This action may be an exploratory action or the greedy action. After taking action a_t and receiving reward r_{t+1}, we arrive in state s_{t+1}. We can define an equation similar to Equation 2.3 for the approximate action-value function:

$$Q(s_t, a_t) \approx r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{2.4}$$

```
Q-LEARNING(s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t)
    if taking the greedy action
        choose a_t to maximize Q(s_t, a_t)
    else
        choose a_t randomly
    if t > 0
        TD_error = r_t + γ max_{a_t} Q(s_t, a_t) − Q(s_{t-1}, a_{t-1})
        Q(s_{t-1}, a_{t-1}) = Q(s_{t-1}, a_{t-1}) + α TD_error
    return a_t
```

Figure 2.10: The Q-Learning algorithm to find the next action.

Unlike Equation 2.3, this is only an approximation. Subtracting the left-hand side from the right-hand side yields the TD error:

$$TD_{\text{error}} = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{2.5}$$

Effectively, the TD error gives the inconsistency between the two approximations. In order to improve the approximation, we want to reduce the inconsistency. If we assume the most recent approximation, $Q(s_{t+1}, a_{t+1})$, is correct we can adjust the first approximation based on the TD error:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha TD_{\text{error}} \tag{2.6}$$

As was the case for neural network updates, a learning rate, α, is used to moderate changes to the function. Q-Learning is performed over many iterations, and giving too much weight to the TD error would likely result in undesired oscillations. A low learning rate stabilizes the learning algorithm and allows better solutions to be reached. The learning rate can be varied over time to improve learning. The complete algorithm is in Figure 2.10.

The major innovation of Q-Learning is off-policy learning. When the TD error is found, the greedy action is used in the computation. However, when the next action is selected we may decide to take an exploratory action instead. Computing the TD error based on the greedy action allows Q-Learning to converge to an optimal policy even though it takes exploratory actions. In this way, Q-Learning can learn the optimal policy while using an ε-greedy policy.

Chapter 3

Mobile Robot Localization

As a mobile robot traverses its environment, it is usually necessary for the robot to maintain a concept of its current location. Depending on the task to be completed, the level of detail may vary. A delivery robot probably needs to know its location in a global coordinate system, while a floor sweeping robot may only need to understand the world directly around it. In both cases, the robots are solving some version of the localization problem.

Typically, roboticists have attempted to localize their robots using map-based representations. Thrun (2002) provides a thorough overview of conventional approaches. While these conventional approaches are map-based, localization as discussed here is independent of the representation used. Any robot that is capable of determining its position, whether absolutely or relatively, is performing localization. With this understanding, a map-based representation is not a precondition for localization. A robot can perform localization with respect to its own understanding of the world, without needing that understanding to be an explicit map.

3.1 Localization Strategies

Before describing the localization problem in detail, it is helpful to describe representations that can be used for describing the location of a robot. In choosing a representation, a roboticist is constraining the localization system in several ways. A robot that moves in two-dimensions can represent its location with (x, y) coordinates on a two-dimensional map, but this is clearly insufficient for a robot that moves in three-dimensions. Likewise, a robot with only laser range-finders cannot use a map made up of photographic images.

It is possible to define two distinct strategies for localization. The first, explicit localization, uses a representation designed in advance by the roboticist. The robot is then responsible for localizing itself with respect to the chosen representation. The second strategy, implicit localization, does not rely on a fixed representation. A robot can have a concept of where it is without being able to specify its location in a predefined representation.

3.1.1 Explicit Localization

In explicit localization, a robot maintains a concept of its current location in a predefined representation. Robots capable of performing complex localization tasks have usually followed this localization strategy. Using a known representation provides several benefits, as discussed hereafter.

Designing a system with known inputs and known outputs is much more straightforward than designing a system that uses an unknown representation. An explicit localization system can easily be tested because there is always a right answer for

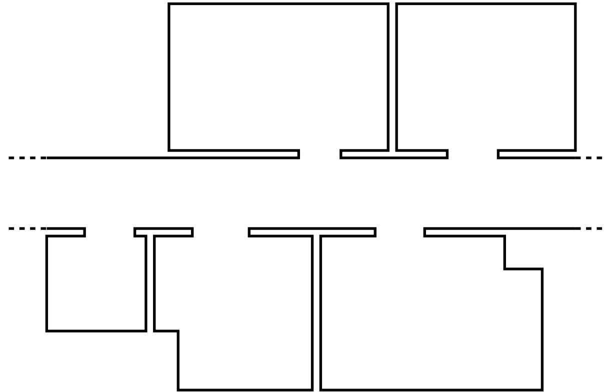


Figure 3.1: An example of a two-dimensional metric map.

the system to output. The robot only exists at a single location, if the localization system gives back the wrong location, then it is relatively easy to debug the system and attempt to fix the error.

A robot that performs explicit localization is also easy to send commands. The robot "speaks" the language that its designer used. A robot that localizes in (x, y) coordinates can be told to go to coordinate $(10.0, 5.0)$, and the robot should end up close to the desired location if the localization system works properly and no unforeseen errors occur.

People are accustomed to using maps. Maps are a straightforward representation for the localization problem, and it is natural that people would attempt to build their robots to use maps also. Maps used on robots can be divided into two classes. Metric maps encode absolute distances in a global coordinate system, while topological maps encode distinctive places and the connections between them. The two types of maps are illustrated in Figures 3.1 and 3.2

Metric maps can be thought of as world-centric. The focus of a metric map is accurately placing features in a global coordinate system. The robot is then able

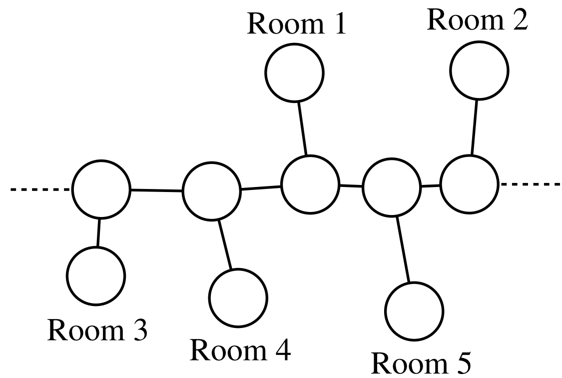


Figure 3.2: An example of a topological map.

to localize by observing features in its environment and comparing against known locations.

On the other hand, topological maps are robot-centric. A topological map encodes observations as *landmarks*, and is able to determine its relative location if it observes a distinctive landmark. If the robot remembers how to navigate from one landmark to another, it is not necessary to use a global coordinate system. The robot can navigate the set of connected landmarks without needing to know exact spatial relationships.

It is also possible for metric and topological maps to be combined. Vaughan et al. (2002) designed a system that allowed robots to travel between places on trails. The presence of distinct places makes the system similar to topological maps, but the trails were described in metric coordinates. The trails only encoded how to travel from place to another. Therefore, unlike a traditional metric map, a trail did not need to be defined in a global coordinate system. Instead, a trail is essentially a set of directions that a robot can follow to move from one place to another.

Metric maps have desirable properties often lacking in topological maps. When a robot builds a metric map of an environment, it creates a representation that tells the robot where it is over the full range of its environment. In contrast, on a topological map the robot only knows about certain distinctive places in the environment and can only move between adjacent places. Therefore, robots with metric maps have a greater degree of freedom to choose how to move through their environment, while topological maps limit the available paths.

3.1.2 Implicit Localization

Animals have amazing localization abilities. Some animals routinely travel for hundreds of kilometers and are capable of returning to known locations. In contrast, robots have somewhat limited abilities. Robots that localize themselves on internal maps are only able to explore areas on the scale of a few kilometers.

The Global Positioning System (GPS) is a common solution to this problem for robots that travel large distances in outdoor environments. Using GPS allows a robot to know its exact location almost anywhere in the world. While GPS is incredibly useful, it is not universally available. Robots that work indoors or underwater are not able to receive GPS signals. While GPS provides a robot with global information, more common sensors provide only local information. For example, laser rangefinders, sonars, and cameras are common on robots. We are interested in using these types of sensors instead of sensors such as GPS that provide a "quick-fix" localization strategy.

Explicit localization has been shown to work very well in a large number of cases. However, examining other possible localization strategies may also be worthwhile.

When using a metric map, a robot can be given a command such as goto (x, y). Specifying commands in this format is only easy for a person who has access to the same metric map. To a lesser extent, specifying commands based on a topological map, such as goto *Room 1*, still require that the person know the locations and names of the nodes in the map. Ideally, we would like a robot that understands commands in human terms. Map-based representations do not directly allow this.

If the task was to interact with people in an unstructured environment, the robot would need to interpret human commands. In such a case the representation must be somehow compatible with the representations that people use. On the other hand, if the task was to drive down a highway the representation could be completely different. In general, there is no single best representation. Instead, the best representation for a task depends on the nature of the task. Using a predefined and explicit representation forces a robot to use a representation that may not be specialized for its current task.

The idea of implicit localization is to remove the dependence on a map, and let a robot use whatever representation allows the robot to accomplish its specified task. In implicit localization, a robot maintains a concept of its current location without explicitly localizing itself on a map. The robot must have some form of internal representation, but the exact form of the representation is flexible and task-dependent. The only role of the representation is to allow the robot to perform its task, so the representation does not need to be interpreted outside of the robot. It is even possible for the robot to choose its own representation. Such a representation can be extremely well suited to a particular task.

The choice of representation is of primary importance to implicit localization. In order to create a system capable of implicit localization, we will design a system that builds an internal representation of a task. If the task requires the robot to maneuver through its environment and perform actions in specific places, then a representation of the task must include the ability to recognize those places. The ability to recognize places demonstrates implicit localization.

Implicit localization is similar to localization on a topological map. When using a topological map, the robot can be localized only at a node on the map. For example, if a robot has a topological map that only has nodes on the perimeter of the room, then it can not accurately localize itself in the interior of the room. Likewise, an implicit localization system is only capable of localizing the robot while it is performing the task that the representation was designed for. Because of this, implicit localization is much more limited than explicit localization on a metric map.

The combination of machine learning and implicit localizations holds promise. A robot should be able to learn a localization strategy that is specially designed for whatever task the robot is performing. There has already been at least one success in this area; Nolfi and Tani (1999) demonstrated a robot that used a neural network representation to implicitly localize itself as it followed the walls around a simple environment.

3.2 Localization Problems

The localization problem can be broken into three subproblems: position tracking, global localization, and simultaneous localization and mapping. These subproblems

were formulated for localization on map-based representations, but they can be easily generalized for localization in arbitrary representations.

3.2.1 Position Tracking

Given a known initial position and a representation of its environment, a robot performing position tracking is able to update its known position as it moves through its environment. Position tracking is the simplest of localization problems, and any localization method must be able to perform this task.

A robot that estimates its position using odometry information is performing a very simple form of position tracking. Odometry information is generally not accurate enough to be useful over long distances. Another method is needed in order to constrain the maximum amount of error that will occur over time. Other sensors such as sonars or laser range-finders usually perform this role.

Robots that only perform position tracking can be unable to recover from large localization errors. If the estimated position is sufficiently different from the actual position, the robot will receive contradictory sensory information and be unable to re-localize itself correctly. Such an occurrence would be fatal to a robot that could only perform position tracking.

3.2.2 Global Localization

Global localization is the ability of a robot to find its location on a map starting from a state in which the robot has no knowledge of its current position. This is sometimes referred to as the "kidnapped robot" problem. Position tracking and

global localization are complimentary problems that are often solved by a single system.

Monte Carlo localization (Dellaert et al., 1999) is a well-known algorithm that performs both tasks concurrently on a metric map. Monte Carlo localization is a probabilistic method that uses a particle filter to estimate the current robot location. The distribution of particles indicates a hypothesis for the location of the robot. There may be a single tight cluster of points, which indicates the robot knows where it is; there may be two or more clusters, which indicates the robot thinks it could be in several different locations; or the particles may be distributed evenly across the environment, which indicates that the robot has no idea where it is. Depending on the distribution, the algorithm can scale from position tracking to global localization.

3.2.3 Simultaneous Localization and Mapping

The last localization problem is Simultaneous Localization and Mapping, or SLAM. In the SLAM problem, a robot does not have a representation of its environment in advance. Through exploration the system must build a representation of its environment. SLAM is generally considered the most important localization problem, and it is also the most difficult. Truly autonomous robots must have the ability to explore and learn to navigate unknown areas.

When mapping a new environment, a robot will usually revisit states that have already been mapped. It is important that the robot is able to recognize when it is revisiting a state, and not add redundant or incorrect information to the map. This requires position tracking when moving in the established map, and global

localization when finding new routes to already mapped locations. Therefore, a robust localization system must handle all three subproblems.

Successful approaches to the problem have been developed using both metric and topological maps (e.g. Eliazar and Parr (2005); Montemerlo et al. (2003); Davison (2003); Duckett et al. (2002)). These works have utilized either range-finding sensors or machine vision. The best of these approaches can build maps at the kilometer scale.

3.2.4 Generalizations to Implicit Localization

In implicit localization, the map is replaced by an unknown internal representation. In this case, position tracking and global localization cannot be viewed as pinpointing the robot location on a map, but should be viewed as minimizing the current confusion of the robot. During position tracking, the internal representation should change predictably. When the internal representation changes unpredictably the robot is confused, and global localization is necessary in order realign the internal representation with reality.

The concept of SLAM is somewhat harder to define in implicit terms. The primary goal behind SLAM is not usually to get a map back from the robot. Instead, the goal is for the robot to understand its environment well enough that it can interact with its environment in an intelligent manner. The key reasons for SLAM are more concerned with *autonomy* than map building. A robot should be able to understand its environment without needing help from people. If it could do that, few people would care if it used implicit localization instead of a map-based representation.

A system that can build representations capable of implicit localization can be used to lessen the burden placed on a robot developer. Where explicit localization algorithms often rely on specific sensors and types of environments, a system that can design its own representation should be able to use a wide variety of different sensors and operate in a wide variety of environments.

The combination of machine learning and implicit localization promises more general solutions to the problem of mobile robot localization. While it is doubtful that a robot would learn a localization strategy on the level of a well-designed algorithm written for a specific environment, it is hoped that a robot can learn "well enough" to complete an intended task.

Chapter 4

Developing Implicit Localization

A robot that could learn how to perform localization would be much more flexible than a robot painstakingly programmed to localize itself in a single type of environment. This chapter describes the concept of developmental robotics and describes how the same ideas can be used specifically for localization.

4.1 Developmental Robotics

Conventionally, robots are designed and programmed to perform a specific task in a specific environment. It is the job of the human designers and programmers to break down the task and find ways to perform each necessary component. This process can be difficult, but is acceptable in many situations. For example, a manipulator arm in a factory can successfully be built to perform its task very well.

In less controlled environments, designing a robot that can perform well is more difficult. The goal of developmental robotics is to allow robots to go through a developmental process much like young children. Starting from a tiny amount of knowledge, a human infant is able to acquire new knowledge from its environment.

If a robot could perform a similar process, then the robot designer would only need to build in simple knowledge. Lungarella et al. (2003) give an overview of development in both humans and robots.

A robot should be able to understand its sensor data better than a person can. While it is natural for people to talk to each other in high level terms, it is probably not appropriate to attempt to talk to a robot in the same way. For example, people understand concepts like hallways, rooms, and windows. When designing a topological map, it is natural to create nodes for such areas. However, those locations might not be distinguishable by a robot with a limited number of sensors. People designing robots are so connected with how they experience the world that they cannot fully understand how a robot perceives its world. Blank et al. (2005) call this *anthropomorphic bias*. By letting a robot develop its own understanding of the world, it will understand and interact with the world in the best way it can. It is no longer forced to do things in the way a person would.

While developmental processes in animals are complex, we will isolate and use only two important concepts: *abstraction* and *anticipation*. Nolfi and Tani (1999) and Blank et al. (2005, 2002) recognized and made use of these two concepts in their works. Although the implementations used by Nolfi, Tani and Blank were quite different, the concepts in use are the same.

Sensor input on mobile robots can be high-dimensional, and working with raw sensor data is often impractical. In order to reduce the amount of data, a developmental process uses abstraction. Abstraction reduces the total amount of data while retaining the important pieces of information. A good abstraction can dramatically

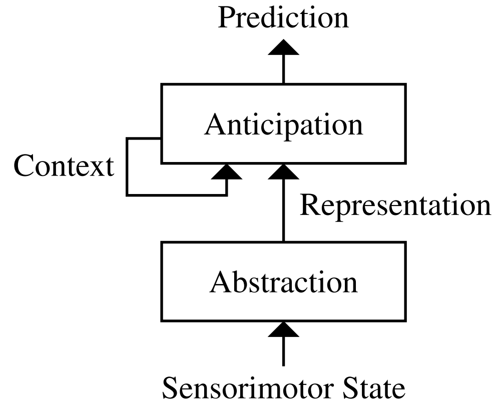


Figure 4.1: A simple developmental system.

simplify the data and make it much easier for other processes to efficiently make use of the important information.

The second important concept is anticipation. While abstraction reduces the total amount of information to consider, anticipation learns about sequences of data. Given a current state and an action, anticipation attempts to predict the outcome. Being able to correctly anticipate the results of an action shows an understanding of the system dynamics. If system dynamics are sufficiently understood, then it is possible for the robot to take actions so as to cause a desired state to occur. In this way, anticipation builds a *model* that a robot can use to understand its interactions with the environment.

While abstraction can be performed with no knowledge of the past, correctly predicting the future often requires previous knowledge. If a robot is attempting to drive to a blue ball outside its current field of view, it must have some memory of where the ball was last seen. Without such knowledge it is impossible to predict where the ball is now.

A simplified version of the system used by Nolfi and Tani (1999) and Blank et al. (2005) is shown in Figure 4.1. Abstraction and anticipation are joined into a multi-layer system, where higher layers receive input from lower layers. First, the abstraction layer receives the sensorimotor information from the robot and forms an abstract representation based on the input. Then, the anticipation layer uses the abstract representation and the current context to produce a prediction and a new context. The current context is the only form of memory in the system. The prediction output must predict something about the future, but what exactly to predict is not of utmost importance. Nolfi and Tani predicted the next abstraction, while Blank predicted the next motor command.

Both Nolfi and Tani (1999) and Blank et al. (2005) claim that multiple layers of abstraction and anticipation can be combined, but neither demonstrate the use of such a system or describe it in great detail. Instead, they chose tasks simple enough that one layer of abstraction and one layer of anticipation could complete the required task. We take a similar approach in this work, but we also attempt to increase the representational power of the lower layers.

The major benefit of multiple layers is that each layer can run at a different timescale. The first layer can make predictions every second while higher layers can predict how lower level representations will change. This temporal abstraction can allow for the representation of longer and more complex behaviors.

However, this benefit is also a drawback. It is not obvious how a layer that predicts events occurring far apart can provide any direct influence over the layers

below. As you go into higher layers, information is continually compressed, in order to go back down this information must be decompressed into real-time action sequences, and it is not clear how to perform this action.

Reinforcement learning with *Options* (Sutton et al., 1999) also makes use of temporal abstractions. An option describes a temporally extended sequence of actions, and by using options it is possible to more efficiently learn tasks when multiple subgoals are important. McGovern (2002) demonstrated the automatic creation of such temporal abstractions. Options are useful for performing behaviors, but they do not address the recognition of known behaviors. In contrast, the layered approach by Nolfi and Tani and Blank et al. is designed for prediction, and it is necessary to recognize what is currently being done in order to make accurate predictions. Recognition is a key component of implicit localization.

It is not immediately obvious that abstraction and anticipation can be used for implicit localization. As a robot moves through its environment, a constant stream of sensor information and motor commands is generated. In order for a robot to understand its relationship with the environment, it must understand the flood of information. Abstraction and anticipation are the tools that will allow this understanding to be developed.

4.2 Bootstrapping Learning

When a developmental robot first begins operation, it has very little knowledge of itself and its environment. In order to train such a robot, it must have some way

to gather initial training data. The training data should come from purposeful interactions with the environment. If the training data were obtained from random interactions, it is unlikely that the anticipation layer would extract any useful behavior.

There are two approaches to building the training data. The robot can be controlled with an innate behavior that is programmed into the robot, or the robot can be controlled by a person during training. Both methods have advantages. If an innate behavior is used, training can continue unsupervised for long periods of time. It is not necessary for a person to guide the actions of the robot during training. On the other hand, the innate behavior must be programmed. Depending on the complexity of the action, it can be difficult to program the innate behavior. Also, it is possible to make use of both types of training examples in a single system (e.g. McGovern (2002)).

In this thesis we will attempt to learn from human examples only. This will avoid the need to write an innate behavior. We want to learn localization from the ground up. If we start from a working algorithm for localization, it is much less clear that anything of value is being learned.

4.3 Learning Abstraction

Abstraction creates a compressed form of raw input. Self-organizing maps (see Section 2.1.3) are one method of creating abstractions. Blank et al. (2005) used self-organizing maps to form abstractions in their work. Given an input sensorimotor state, a single node in the self-organizing map will be activated. This node represents

an abstraction of the input. From the active node it is even possible to get a representative sensorimotor state.

Furthermore, if only the sensor information is used in the matching, it is possible to use a self-organizing map as a lookup table when deciding what the motor commands should be. An obstacle avoiding robot has been constructed using only a single abstraction layer (Blank et al., 2002). Such a system is incapable of learning anything but the simplest behaviors. The system has no memory, so only purely reactive behaviors can be learned in this way.

Nolfi and Tani (1999) took a somewhat different approach. The developmental model used by Nolfi and Tani only attempted to "understand" the environment. There was no attempt to use the learned model for control. Therefore, it was not important to provide a direct mapping from input states to control actions. So, even the abstraction layer could operate on sequences of data.

The abstraction layer was composed of two neural networks. The first was an Elman network (see Section 2.1.2) trained to produce the next sensorimotor state given the current sensorimotor state. The robot had eight infrared sensors and two motors, so the combined sensorimotor state consisted of 10 values: eight ranges and two motor speeds. The network had a three node hidden layer. The hidden layer node activations were provided as input to the second neural network. This network had no hidden layer and three outputs. Each output node had a recurrent connection to itself. After both neural networks were evaluated for the current sensorimotor state, the output node of the second network with the highest activation was set to 1, while the other two were set to 0. Therefore, the layer only had three distinct outputs.

The design used by Nolfi and Tani (1999) performed a very similar function to a self-organizing map. The final abstraction recognized walls, hallways, and corners. (Blank et al., 2002) recognized the same types of inputs with a single SOM. This is possible because these concepts are not dependent on the fact that they were obtained from a prediction network, but are observable in the data at any individual timestep.

Robots that only explore simple environments can require SOMs with a large number of nodes. This is because SOMs provide a coarse distinction between nearby states. If the robot needs to take a different action depending on how far it is from the wall, each distance must have a corresponding node in the SOM. Still, though, as the robot moves it passes harsh transitions from one active node to another. If the raw input changes continuously, usually the nodes will be somewhat near each other in the network topology, but that is not necessarily true.

The world is continuous, and forcing a discrete representation seems unnecessarily constraining. Therefore, we will use an abstraction mechanism similar to Nolfi and Tani (1999), but with three differences. First, we will use only a single network and not discretize the output. Instead, we prefer to use a continuous abstraction that recognizes fine distinctions between states. Second, we will not use a recurrent network. The recurrent network used by Nolfi and Tani was not shown to perform a useful function, and they reported similar performance when recurrent connections were not used. Third, we will train the network to output the current sensor values instead of the next sensor values. The system used by Nolfi and Tani only recognized concepts that are apparent from a single set of sensor readings, so it is unclear

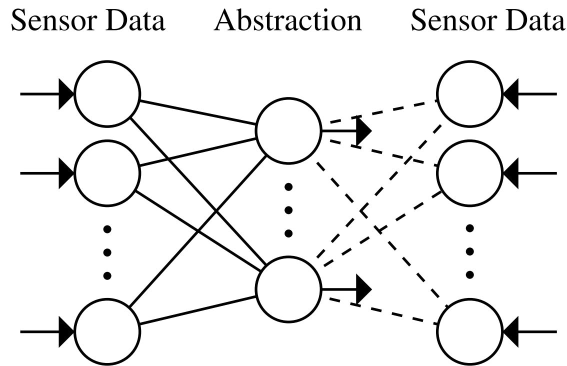


Figure 4.2: The proposed abstraction layer for learning localization. The hidden layer representation is provided as an abstract representation of the input. The output layer is used only during training, and can be removed afterward.

that using prediction here is necessary. This change allows the abstraction and anticipation to be clearly separated in our system.

Motor commands will not be present in the abstraction network. After training is completed, the output layer is removed, and the hidden values are used as an abstract representation of the inputs. The only role of such a network is data compression. High dimensional sensor input is converted to a lower dimensional representation. An example network is shown in Figure 4.2. This is a very simple example of an auto-associative neural network. See Kramer (1991) for more information about the uses of such networks.

Sensors usually return redundant information. For example, if a single pixel on an image returned from a camera is lost, it is usually possible to fill in the empty pixel to a high degree of accuracy. Even if every other pixel is missing, a person can still understand the content of the picture. Being able to compress sensor data reflects an understanding of the relationships between individual readings, and is a reasonable goal for an abstraction layer.

If the training data encompasses all situations which the robot will encounter, sensors using different modalities can be combined in a single abstraction network. If a robot was trained in an environment with small blue rooms and large red rooms, it would learn to associate red vision inputs to large range readings. In general though, it is probably better to use a different abstraction network for each sensor modality. Data compression within a single sensor modality is more likely to learn useful abstractions. Compressing unrelated sensor data is more likely to find coincidental occurrences. This is one reason that motor commands will not be used as input for the abstraction network.

4.4 Learning Anticipation

Anticipation is somewhat more difficult to learn than abstraction. The need to retain a memory of previously observed data necessitates the use of an algorithm that produces both a prediction and an updated context (see Figure 4.1). Recurrent neural networks are a common solution to this problem (Nolfi and Tani, 1999; Blank et al., 2005, 2002), and will also be used here.

Our goal is to demonstrate implicit localization, so the anticipation layer must be designed to provide a representation of the robot's next location given its current location. We want our system to provide information that is useful for control. Having a concept of location is important, but if the location information cannot be used effectively by the robot control system, then it is of little value.

Both Nolfi and Tani (1999) and Blank et al. (2005) used Elman networks to anticipate the future. However, because both sets of authors used discrete abstractions,

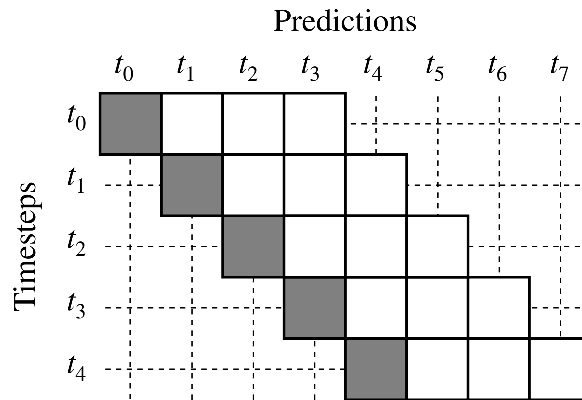


Figure 4.3: A cascading sequence of predictions from a three-step prediction network.

they could run their anticipation networks whenever the abstract representation changed. This allowed the networks to be evaluated much less frequently, and to remember longer sequences of data. However, if we have an abstract representation that changes at each timestep, then we have to make predictions at each timestep. Therefore, the sequence of data that we are attempting to predict is much longer and more complex.

As discussed in Section 4.1, layers of abstraction and anticipation can be used to build an "understanding" of complex sequences of data, but it is unclear how higher layers can be used to directly influence the actions of the lower layers. Therefore, we propose to extend the power of a single anticipation layer to recognize sequences that are more complicated than current methods have allowed. In order to do this we will replace the traditional single-step anticipation layer with a multi-step anticipation layer.

Like Nolfi, Tani, and Blank, we will use an Elman network for anticipation. Our multi-step anticipation network will predict not only the next abstract representation, but the next *sequence* of abstract representations. For example, at timestep t we generate predictions for timesteps $t+1$, $t+2$, $\cdots$, $t+n$, and at timestep $t+1$ we generate predictions for $t+2$, $t+3$, $\cdots$, $t+n+1$. An example sequence of predictions is shown in Figure 4.3.

The complete abstraction and anticipation system is shown in Figure 4.4. The anticipation network takes as input the representation from the abstraction layer and the current motor commands and velocities. It provides n predictions as output. The anticipation network outputs sensor states in the abstract representation formed by the abstraction network. If it is necessary to reconstruct the real sensor values, the hidden and output layers of the abstraction network can be used to decompress the information stored in the abstraction representations.

An n-step prediction network can be trained in batch mode using sensor logs from tasks performed under human control. The network is trained sequentially across a sensor log. Initially, the representation and motor commands from the first timestep are input into the recurrent network, and all the context nodes are set to zero. Each output prediction is compared to the correct output in the sensor log, and backpropagation is used to train the network. The hidden representation is copied to the input layer and the next representation and motor commands are passed through the network. The predictions are once again compared, and backpropagation is used to update the network weights. The process continues until the end of the sensor log is reached. The abstraction training algorithm is shown in Figure 4.5, and the anticipation training algorithm is shown in Figure 4.6. Both figures show a

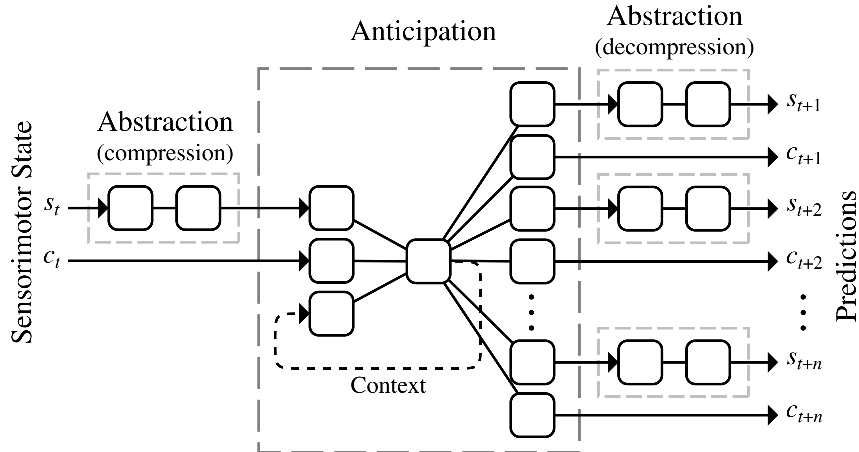


Figure 4.4: The proposed system for learning localization. Each rounded square represents multiple nodes. s_t is the sensor input at time t, and c_t is the motor commands and velocities at the same time. The input and hidden layers of the abstraction network are used to build a compact representation. After the anticipation networks is evaluated, the hidden and output layers of the abstraction network can be used to rebuild the full sensor information.

```
TrainAbstraction(abstraction, s)
  for i = 0 ... length(s) - 1
    abstraction.propagate(s_i)
    abstraction.backpropagate(s_i)
```

Figure 4.5: Pseudo-code for the abstraction training algorithm. s is an array of sensor readings.

single training iteration, but multiple iterations of training are necessary for both abstraction and anticipation. Towards the end of a run, the long-term predictions will not be available in the sensor log. In such a case training proceeds as normal, but when backpropagation is performed no error is attributed to the outputs that correspond to predictions past the limits of the sensor log.

The addition of longer term predictions can improve the performance of a network at short term predictions. If the difference between abstractions at adjacent timestep changes continuously, then it is possible to predict fairly accurately based on only

```
TrainAnticipation(abstraction, anticipation, s, c)
  anticipation.clearHiddenLayer()
  for i = 0 ... length(s) - 2
    abstraction.propagate(s_i)
    r_i = abstraction.getHiddenLayer()
    h_i = anticipation.getHiddenLayer()
    predictions = anticipation.propagate(r_i . c_i . h_i)
    targets = new Array()
    for j = 1 ... predictions.length()
      if i + j < length(s)
        abstraction.propagate(s_{i+j})
        r_{i+j} = abstraction.getHiddenLayer()
        targets .= r_{i+j} . c_{i+j}
    anticipation.backpropagate(targets)
```

Figure 4.6: Pseudo-code for the anticipation training algorithm. Array concatenation is indicated by the '.' operation. s is an array of sensor readings, and c is an array of motor commands and velocities.

local information. In this case only a shallow representation is necessary. All we need to remember is how the abstraction is changing, and we can accurately predict future abstractions. In this case single-step prediction would suffice.

However, if the abstraction usually changes continuously but changes drastically on rare occasions, a single-step prediction network will learn to predict only the continuous changes. It is easier to predict the small changes that usually occur than it is to predict the large changes that occur only rarely. A one-step prediction network has little incentive to learn how to predict the drastic changes, and will instead learn to predict the more common changes very well. On the other hand, a multi-step prediction network has more incentive to learn how to predict the drastic changes, because they are much more common in long-term predictions. The long-term predictions will make use of the hidden layer to store information and improve their performance. After sufficient representation is built on the hidden layer, even

the short term predictions can use the hidden representation in order to improve their accuracy.

This is best demonstrated with an example. We will examine how anticipation can be used to predict an alternating sequence of 10 zeros and 10 ones. By predicting that the next value will match the current value, a one-step prediction network could achieve 90% accuracy. With such high accuracy, it would be difficult for the recurrent network to learn the transitions from one to zero, even though they are the defining events in the sequence. In contrast, if we have a 9-step prediction network it will be much more inclined to learn to predict the transitions. The 5-step prediction can only achieve 50% accuracy by predicting the current value, so it will be more likely to develop useful hidden representations. In turn, the representations developed because of the presence of the 5-step prediction will also be available to the other 8 predictions, and the entire network will learn to predict better than otherwise possible.

Figures 4.7 and 4.8 show the error reduction during the training of 1-step and 9-step prediction networks for the alternating sequence problem described above. The networks have a single input, a four node hidden layer, and an output node for each prediction. The networks are trained with a learning rate of 0.001. In Figure 4.8 there are 9 lines ranging in color from black to light gray. The black line represents the prediction error of the 1-step prediction, while the lightest line represents the 9-step prediction. We see that 1-step and 9-step predictions are initially made with high accuracy, while the predictions near 5-steps are the least accurate. As training progresses, the predictions with high error get more accurate. In turn, once the hidden layer builds adequate representations, the error decreases for

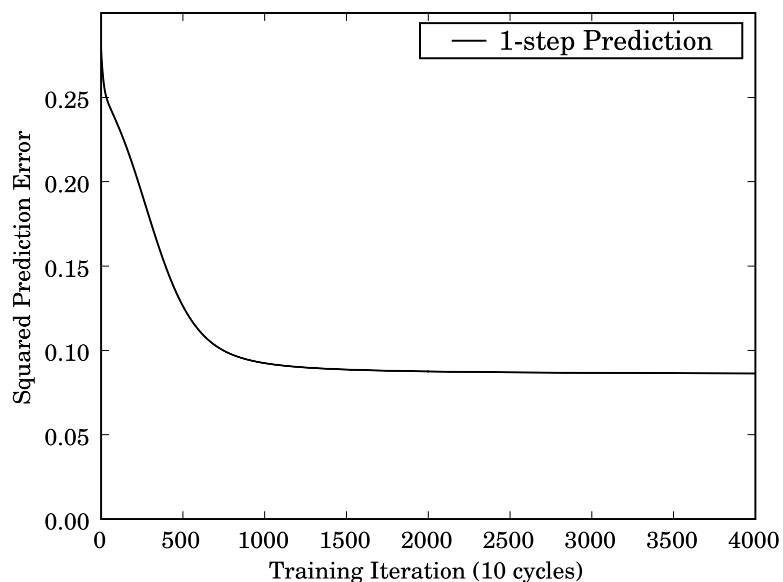


Figure 4.7: Squared prediction error during the training of a 1-step prediction network for an alternating sequence of 10 zeros and 10 ones. Averaged over 30 trials.

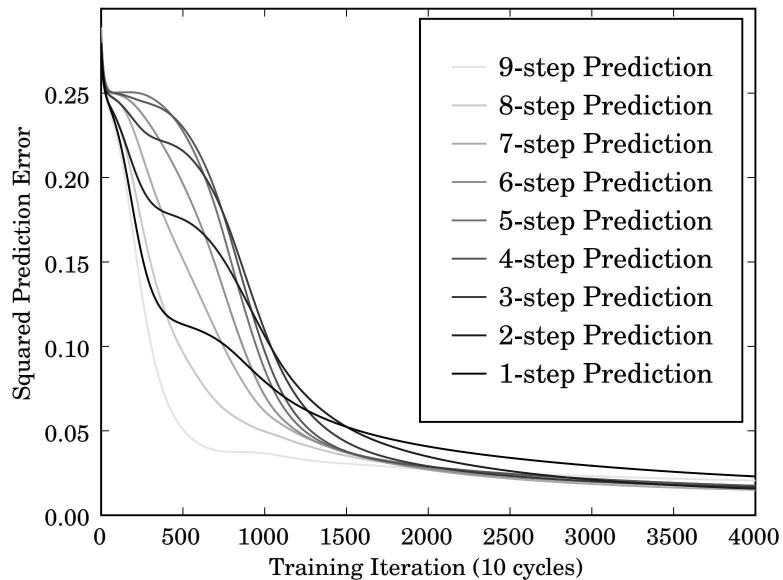


Figure 4.8: Squared prediction errors during the training of a 9-step prediction network for an alternating sequence of 10 zeros and 10 ones. Darker lines indicate shorter term prediction, with the 1-step prediction in black. Averaged over 30 trials.

```
PredictiveControl(robot, abstraction, anticipation, lookahead)
  anticipation.clearHiddenLayer()
  i = 0
  while robot.isRunning()
    // read the current state
    s_i = robot.getSensorState()
    c_i = robot.getMotorCommands()

    // make a new set of predictions
    abstraction.propagate(s_i)
    r_i = abstraction.getHiddenLayer()
    h_i = anticipation.getHiddenLayer()
    p_i = anticipation.propagate(r_i . c_i . h_i)

    // get the first motor command from the last prediction
    commands = getMotorCommands(p_i, 1)
    for j = 1 ... lookahead
      if i − j >= 0
        // add motor commands from previous predictions
        commands += getMotorCommands(p_{i−j}, j + 1)

    // find the average command from all predictions
    commands = commands / min(i + 1, lookahead)

    // execute current action and set next action
    i = robot.advanceTime()
    robot.setMotorCommands(commands)
```

Figure 4.9: Pseudo-code for the predictive control algorithm. Array concatenation is indicated by the '.' operation.

every prediction. In contrast, the 1-step prediction network is able to do only slightly better than 90% accuracy. This simple example shows how long-term predictions can force a recurrent neural network to learn better representations than would otherwise be possible. The 1-step anticipation network has the same amount of representational power as the 9-step, but it never manages to learn a solution as good as the 9-step prediction network.

If an anticipation network is trained until the prediction error becomes very low, it is possible to use the predicted motor commands in place of motor commands provided by the teacher (see Figure 4.9). At this point the network is capable of repeating the task it was trained for. This is the goal of the developmental processes in use here. In contrast, much work in developmental robotics is concerned with emergent behavior. The end goal of developmental robotics is robots that learn about themselves and their environments while developing their own goals. That is clearly not the case here. We are interested only in emergent representations as they can be used for implicit robot localization.

Abstract Hidden Markov Models (AHMMs) (Bui et al., 2002) are another model that can be used to predict events at multiple timescales. Osentoski et al. (2004) used an AHMM to predict what path a person was taking through an environment. As more of the path was seen, the prediction accuracy increased. AHMMs are primarily concerned with activity recognition. Although it may be possible to use a AHMM as a predictive control system, current work is not focused in that direction.

An AHMM can be used to assign probability to an entire set of actions. In contrast, a neural network is somewhat limited because it can only report a single output. In stochastic environments this may severely hurt the performance of an anticipation neural network, while an AHMM can still work very well. In this thesis we will only examine implicit localization in static environments, and it may be necessary to use other models to generalize to dynamic environments.

4.5 Improving Performance

Nolfi and Tani (1999) developed a robot that could localize itself along the path it was trained on. The robot continuously followed the wall around a simple room while predicting what its sensor readings would be in the future. After training, it was shown that the robot could accurately predict future states. In other words, the robot had an implicit representation of its current location that allowed it to predict the future. When the robot was unexpectedly moved, its predictions would not accurately reflect the future, and the *prediction error* would be much higher. The prediction error essentially showed that the robot was confused. However, after a brief amount of time the prediction error would once again shrink to its previous level. When the prediction error returned to a small value, the robot had successfully re-acquired its location in a process similar to global localization.

Prediction error is a very important concept. It tells us how well the robot understands its environment and its current location. Nolfi and Tani (1999) only used the prediction error as an indicator of confusion, but we can also use it to improve performance. If the goal is to perform the task that the network was trained for, then a high prediction error indicates that for some reason the robot is not successfully performing the task. Predictions are only good when the prediction error is low.

We can define prediction error in two ways. We define *offline* prediction error as the difference between a prediction and what really happens, and we define *online* prediction error as the difference between successive predictions. During training,

```
OFFLINEPREDICTIONERROR(i, r, c, p, lookahead)
    e_i = 0

    for j = 1 ... lookahead
        if i - j >= 0
            rError = r_i - getRepresentation(p_{i-j}, j)
            rError = sum(rError * rError)
            cError = c_i - getMotorCommands(p_{i-j}, j)
            cError = sum(cError * cError)
            e_i += (rError + cError) / lookahead

    return e_i
```

Figure 4.10: Pseudo-code for offline prediction error computation. i is the timestep, r is the abstract representation, c is the motor commands and velocities, p is the predictions, and lookahead gives the lookahead length of the anticipation network.

we attempt to minimize the offline prediction error. This is possible when training in batch mode because we know exactly what will happen during the rest of the run.

When the prediction error is high, another mechanism is needed in order to get the robot back on task. Reinforcement learning can be used for this task. The robot can be rewarded when the prediction error is reasonably low. The robot will learn to perform the task as closely as possible in order to receive the maximum amount of reward. Designing a good reward function for reinforcement learning is a non-trivial process, so automatically generating a reward function is advantageous.

In addition, such a reward function is fully self-contained in the robot. No external information must be provided. It is common to use global information for reinforcement signals. For example, goal finding robots often use the distance from the goal as a penalty or reward (Provost et al., 2006; Smart and Kaelbling, 2002). When using reinforcement learning in such a way, it is necessary to have a system in place that can provide the necessary global information to the agent. By using the

```
OnlinePredictionError(i, r, c, p, lookahead)
   e_i = 0

   if i == 0
      return e_i

   rError = r_i - getRepresentation(p_{i-1}, 1)
   rError = sum(rError * rError)
   cError = c_i - getMotorCommand(p_{i-1}, 1)
   cError = sum(cError * cError)
   e_i += (rError + cError) / lookahead

   for j = 1 ... predictions - 1
      rError = getRepresentation(p_i, j) - getRepresentation(p_{i-1}, j + 1)
      rError = sum(rError * rError)
      cError = getMotorCommand(p_i, j) - getMotorCommand(p_{i-1}, j + 1)
      cError = sum(cError * cError)
      e_i += (rError + cError) / lookahead

   return e_i
```

Figure 4.11: Pseudo-code for online prediction error computation. i is the timestep, r is the abstract representation, c is the motor commands and velocities, p is the predictions, and lookahead gives the lookahead length of the anticipation network.

prediction error as a reward, we can automatically create a reward function using only local information and avoid the need for another system.

The idea of using a level of certainty to generate a reward signal is not new. Barto and Şimşek (2005) rewarded a reinforcement learning agent for untargeted yet novel behavior. The agent learned about its environment much more quickly because it received reward for doing new things. Essentially, the agent had an internal curiosity that encouraged it to more fully interact with its environment. Recently, Şimşek and Barto (2006) have used the difference between successive approximations of a value function as a reward signal. When the value function approximation changes, it indicates that the agent does not understand its interactions with its environment. By rewarding the agent for visiting states that it does not understand, it is forced to more fully explore that environment. We are proposing the inverse process here, where an agent is rewarded for doing things it understands well, but it would also be possible to use an anticipation network to reward an agent for behavior that caused high prediction error.

4.6 Application to Localization

The developmental processes of abstraction and anticipation are not specifically designed for localization, but they can be used for localization. An anticipation network learns to predict what will happen only in situations that are similar to what it was trained on. So, the prediction error tells us how well the robot "understands" what it is currently doing. In turn, we can use the prediction error to get an idea of the location of the robot.

In the previous sections, we discussed using abstraction and anticipation to understand the sequence of sensorimotor information generated by a robot performing a specific task. Abstraction and anticipation are task independent, and almost any task could be chosen. If the chosen task requires a robot to move through a complex environment, then a successful anticipation system will also exhibit properties of implicit localization. In order to predict future sensor inputs, the robot must have an idea of where it currently is. This information is stored in the weights of the network itself.

While it is not possible to extract the localization component of the system, it is possible to make use of it. For example, five anticipation networks could be trained on five different paths through an environment. A person could then drive the robot through the environment on a similar path while each anticipation network attempted to predict future states. The prediction error from each of these networks should then tell us how close the robot is to each of the initial paths. This can also be examined on a single path. If prediction error is usually high, then the robot is not on the path, but if prediction error is usually low, then the robot is most likely on a similar path to the one it drove during training.

If the five paths in Figure 4.12 were used, then it should be possible to use the prediction errors to determine which path was followed most closely. By knowing which path was followed, we can also determine which room the robot entered. This demonstrates coarse-grained localization. However, finer-grained localization should occur inside each network. The path that leads to *Room 5* passes three doorways before turning into the fourth. An anticipation network trained to predict this path must recognize when each doorway is passed, but not turn until the fourth doorway

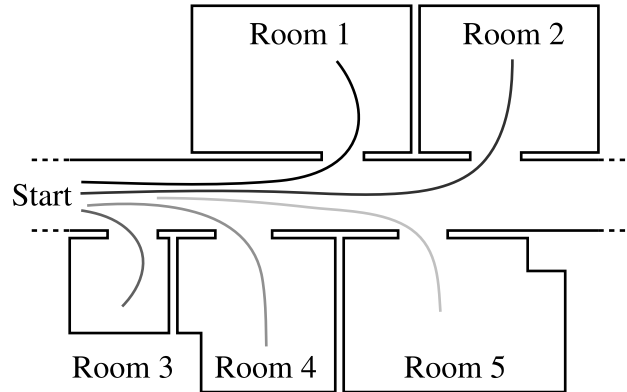


Figure 4.12: Five paths through an indoor environment.

is seen. Alternatively, the network could learn to take the first doorway on the right after the first doorway on the left. Either approach is valid, and the anticipation network can choose to learn whatever is the easiest for it.

The use of multi-step prediction also forces the network to learn more information. Instead of only learning when to turn, the network also has to predict turns far in advance. Therefore, the network not only knows to turn into the third door on the right, but it also knows when it is going to pass the first and second doors. To accurately predict all this information, it must have an expressive representation of the task. As the robot drives, it is localizing itself with respect to its own representation. If a network can be trained to do this, it is performing implicit localization.

The system proposed in this chapter is related to several other systems, but it has components that make it better for implicit localization than any of the others. Reinforcement learning with temporally extended actions, or options, is good for learning how to perform sequences of behavior, but it does not learn how to recognize them. Similarly, Abstract Hidden Markov Models are good at recognizing behavior, but have not been used to control a system. When trying to perform

implicit localization, we desire both the ability to recognize places we have traveled before and the ability to use predictive control. Neither options nor AHMMs give us these capabilities in a single package. It may be possible to combine the two systems, but the proposed system from this chapter is much simpler. It is also hoped that it will perform “well enough” to warrant real world applications without needing to add additional complexity.

Chapter 5

Evaluating Implicit Localization

In the previous chapters we have described the machine learning algorithms, the localization problem, and the foundations of development. In this chapter all three systems come together into working systems that can perform localization in simulated worlds.

5.1 Path Following

In order to demonstrate localization, it is sufficient to demonstrate adequate performance at a task that requires localization. The task we chose is *path following*. In order to follow a path, a robot must know where it is and know what action to take. In simple environments, path following can be done purely reactively. Each sensor input can be mapped to a corresponding motor command. Such situations are not of great interest. We will define paths where the input does not uniquely identify the proper action. In order to traverse such a path, it is necessary for the robot to implicitly localize itself along the path.

It is possible to perform localization without being in control of a robot. However, we want to demonstrate the usefulness of implicit localization, so we choose a task that can be used to test both localization and control. Path following is an ideal task for this reason.

5.2 Grid World

The first environment we explore is a grid world in which each cell is associated with three random values such that neighboring cells have values near each other. Such an environment can easily be represented by a bitmap image file in which the red, green, and blue values for each pixel are attributed to the corresponding cell. The values of each pixel in the image are chosen randomly, but by applying a blur to the image nearby pixels are given similar color values. After being smoothed, the image is adjusted so that all colors are still represented. The color values are integers ranging from 0 to 255, but are normalized to be continuous values in the range from 0 to 1.

The robot in this environment can move one cell at a time. The robot is allowed to move in the four cardinal directions. At each timestep the robot receives the random values from its current cell as a sensory input. The robot has three sensory inputs: red, green and blue. Gaussian noise with a standard deviation of 0.05 is added to each sensor reading. After receiving the sensor readings, the robot must choose which direction to travel next. The four actions and their representations are shown in Table 5.1. After an action is selected there is a 95% chance that the robot will move in the specified direction and a 5% chance that the robot will not move.

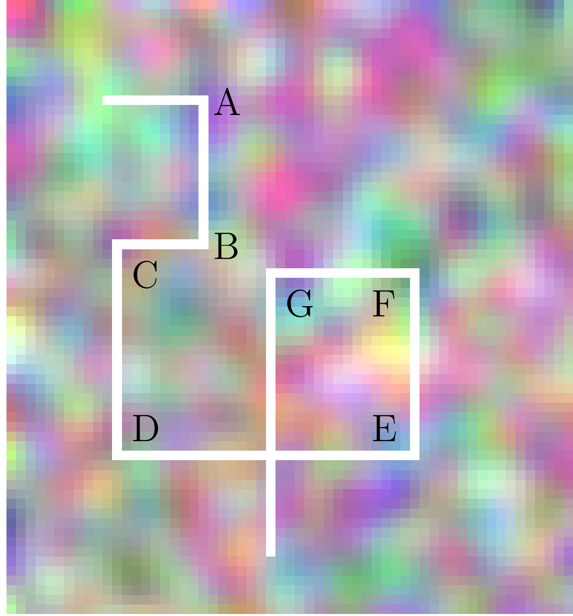


Figure 5.1: The grid world domain and path. The robot starts at the upper left.

	Action	Representation
1.	Up	(1, 0, 0, 0)
2.	Down	(0, 1, 0, 0)
3.	Left	(0, 0, 1, 0)
4.	Right	(0, 0, 0, 1)

Table 5.1: All possible actions in the grid world.

The path that we attempt to teach the robot is shown in Figure 5.1. The path is 150 cells long, but due to missed actions it takes the robot about 158 timesteps on average to traverse the path. The three sensor values are already independent of one another, so there is no need for an abstraction network. Instead, we can focus on anticipation only. Every time the robot drives down this path it will receive slightly different sensor readings and perform actions at slightly different times. Therefore, following the path is more difficult than just memorizing a set of actions.

5.2.1 Anticipation

When attempting to train an anticipation network, there are several important parameters to set. The size of an anticipation network's hidden layer places an upper bound on the complexity of the model that can be represented. For the grid world path following task we used networks with 20, 40, and 80 node hidden layers. We hypothesize that networks with more representational power due to bigger hidden layers will be better able to complete the task.

The next important parameter is the number of future predictions to make. In the previous chapter it was shown that a one-step prediction network failed to learn to predict the transitions in a simple alternating sequence. Making longer term predictions increases the ability of a network to handle sudden changes. In order to evaluate the importance of multi-step prediction we attempted to learn to follow the path using single-step prediction networks and 10-step prediction networks.

The last important parameter is the number of training examples to provide. The grid world used here is not deterministic, every time the robot traverses the path it will record slightly different information. The number of unique sensorimotor logs available for training may greatly affect the ability of the network to generalize. In our testing we used 1, 10, and 100 sensorimotor logs for training.

In addition to the sensorimotor logs used for training, we held out an extra 10 sensor logs to use for testing. These sensorimotor logs were recorded as the robot traversed the same path, but were not available for the network to train on. These logs will be used to determine if a network is overfitting its training data. If overfitting occurs, prediction error will decrease on the training set while increasing on the test set. Overfitting is more likely when there are fewer sensor logs in the

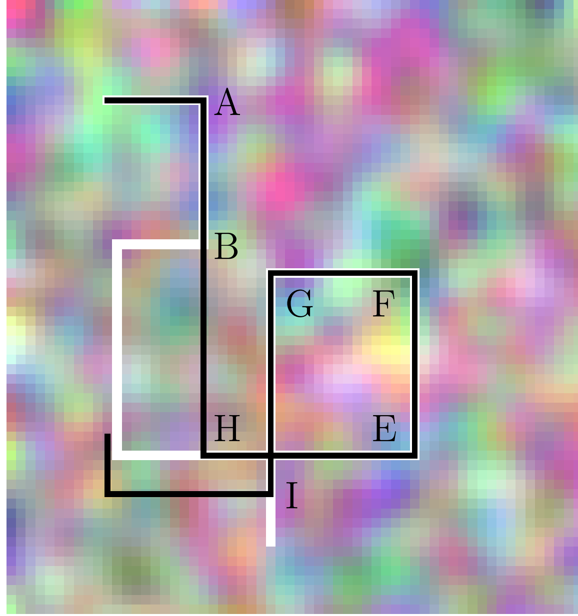


Figure 5.2: An overlapping path through the grid world. The paths overlap for a significant portion of the run. The new path is shown in black, the original path is in white.

training set and when the model used for anticipation is more complex. Therefore, networks with many hidden neurons that are trained with few examples are likely to overfit the training data and perform poorly on the test set. Such networks may not be useful for localization.

Sensor logs were also recorded along two additional paths. The first of these paths had a long region of overlap with the original path (Figure 5.2), while the second is entirely different than the original path (Figure 5.3). We expect prediction error to be significantly lower along the overlapping path, because it is more similar to the original path. 10 sensor logs are recorded along each path. We will call these sets of 10 sensor logs the overlapping set and the different set.

Eighteen sets of thirty experiments were carried out with each possible combination of parameters. All the networks were trained 20,000 times with the anticipation

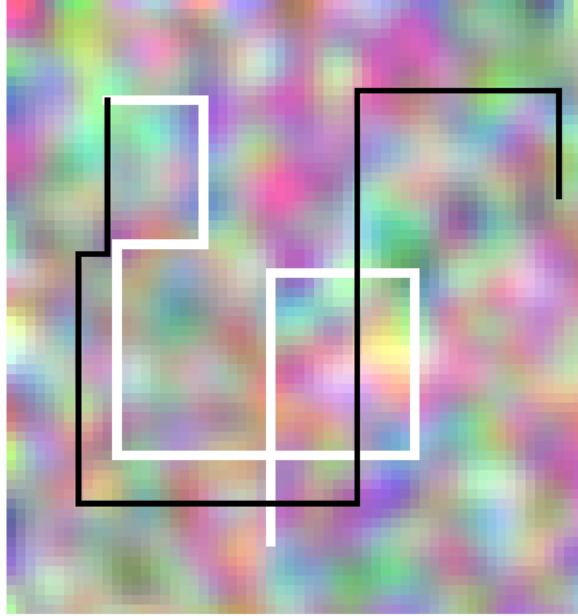


Figure 5.3: A different path through the grid world. Both paths start at the same location, but are otherwise different. The new path is shown in black, the original path is in white.

training algorithm presented in Figure 4.6. Entirely new data is generated for each experiment. When multiple sensor logs were used for training, the sensor log was cycled in order between calls to the TrainAnticipation algorithm. The learning rate was set to 0.001, and momentum was set to 0.5. The sensor inputs were used as an abstract representation, and no abstraction network was used. All nodes in the network used sigmoidal activation functions. Offline prediction error on the training set was computed continuously during training. After every 100 iterations of the TrainAnticipation algorithm, the average offline predication error was calculated for the test set, the overlapping set, and the different set by using the algorithm in Figure 4.10.

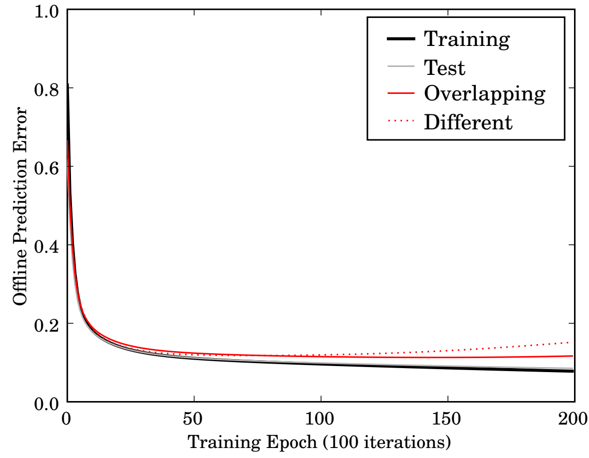
We hypothesize that the single step prediction networks will accurately predict the changing sensory information, but will not accurately predict changes in motor

commands. The motor commands only change seven times during the training run, so it is unlikely that a one step prediction network would learn to predict them. On the contrary, it is hoped that the 10-step prediction networks will be able to better predict the changes. The 10-step prediction network must predict changing motor commands much more often, and should learn to predict them much more accurately.

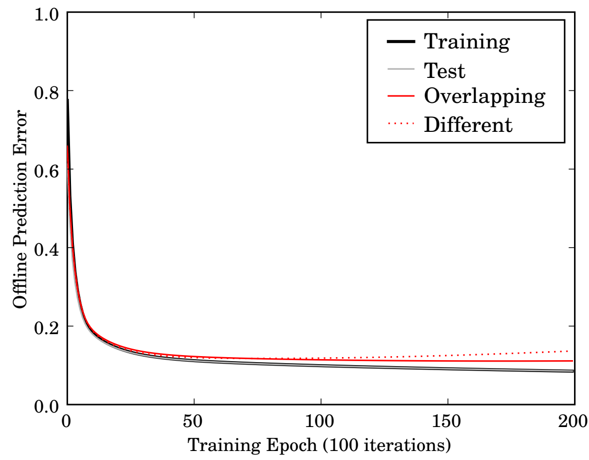
Figures 5.4, 5.5, and 5.6 show the error reduction during training of the 1-step prediction networks. Figures 5.7, 5.8, and 5.9 show the error reduction during training of the 10-step prediction networks.

The one-step prediction networks (Figures 5.4, 5.5, and 5.6) all had similar training results. Prediction error decreased very quickly on all five sets of data. Over time the overlapping and different sets began to receive more error. There was little overfitting here. The networks trained with 1 training example and an 80 node hidden layer had the highest amount of overfitting, but on average the prediction error was still decreasing on the test set even at the end of the experiments.

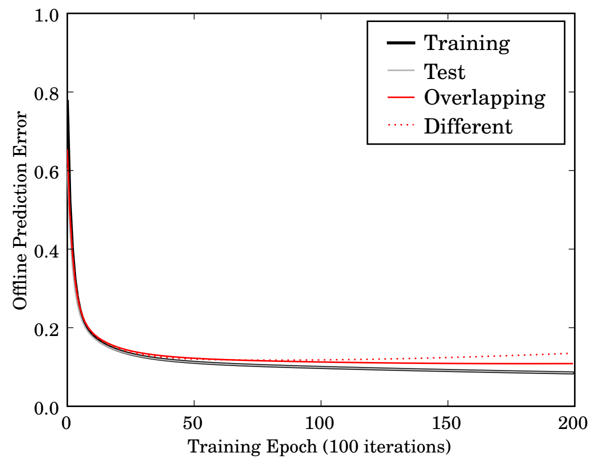
The results from the 10-step prediction networks (Figures 5.7, 5.8, and 5.9) are different than the 1-step prediction networks in several important ways. First, while the prediction error on the overlapping and different sets was low when 1-step prediction was used, it is much higher when 10-step prediction is used. Second, learning occurs at a slower rate than when training a one-step prediction network. Finally, we see that overfitting is much more of a problem when only a single training example is used. There is still very little overfitting when 10 or 100 training examples are provided.



(a) 1 training example

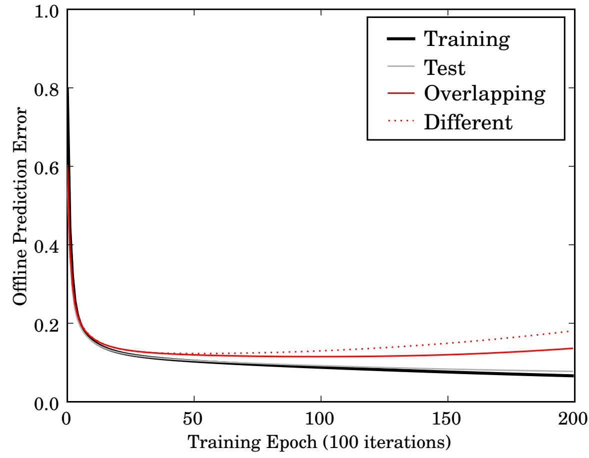


(b) 10 training examples

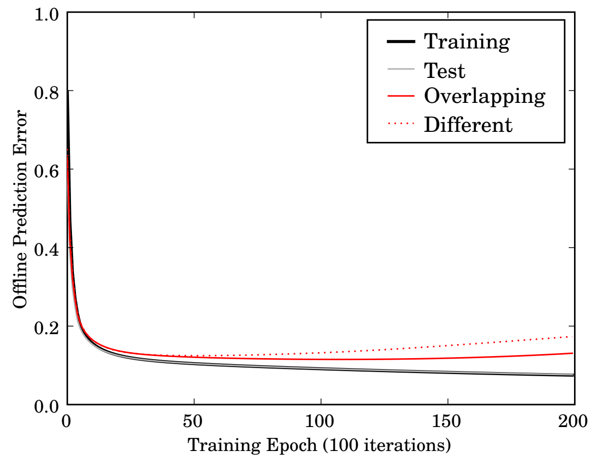


(c) 100 training examples

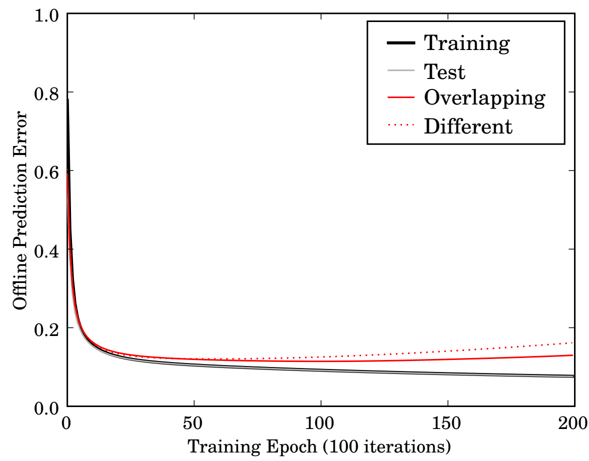
Figure 5.4: Training 1-step prediction networks with 20 hidden nodes and 1, 10, or 100 training examples. Results averaged over 30 experiments.



(a) 1 training example

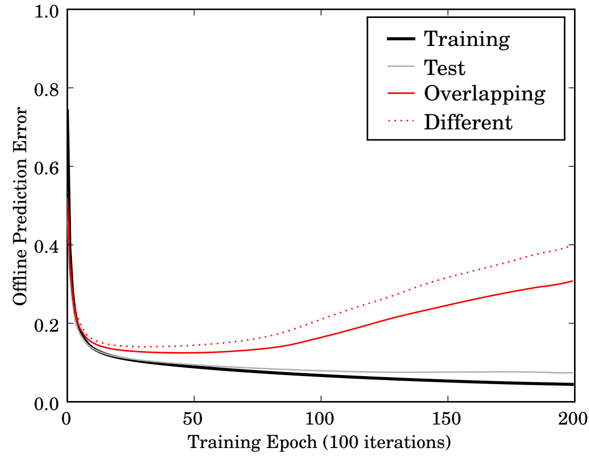


(b) 10 training examples

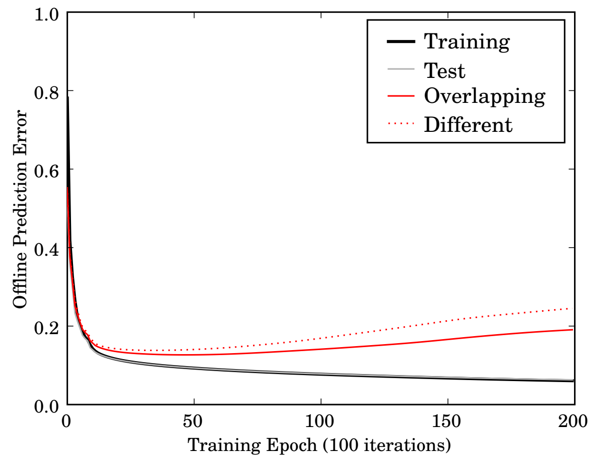


(c) 100 training examples

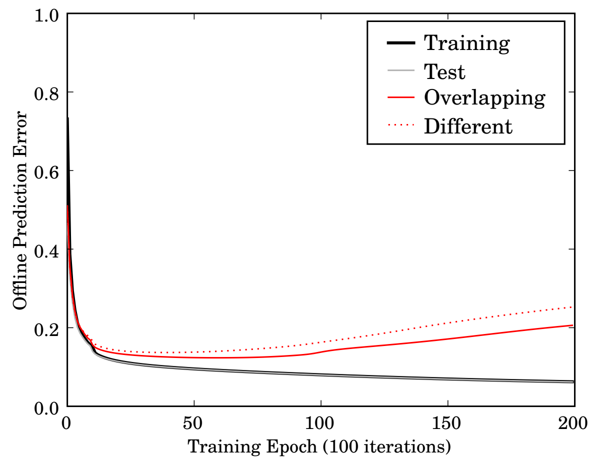
Figure 5.5: Training 1-step prediction networks with 40 hidden nodes and 1, 10, or 100 training examples. Results averaged over 30 experiments.



(a) 1 training example

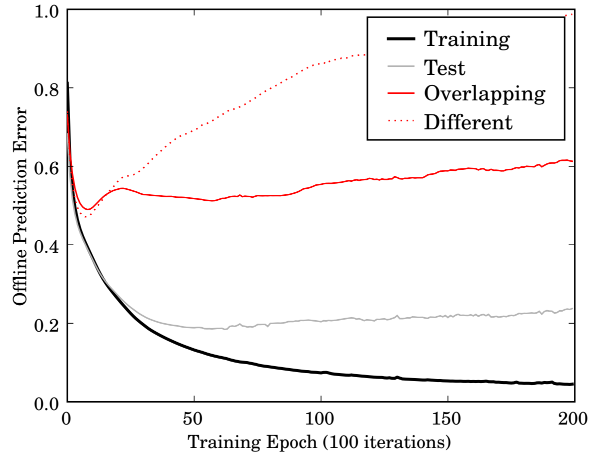


(b) 10 training examples

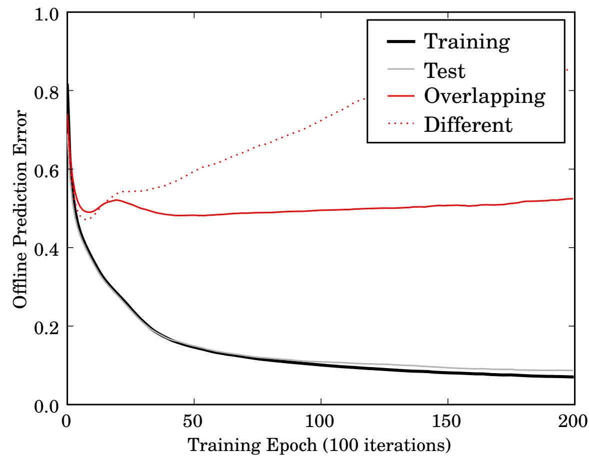


(c) 100 training examples

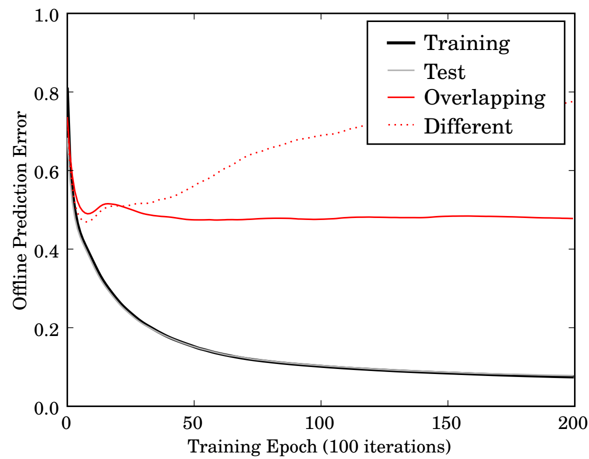
Figure 5.6: Training 1-step prediction networks with 80 hidden nodes and 1, 10, or 100 training examples. Results averaged over 30 experiments.



(a) 1 training example

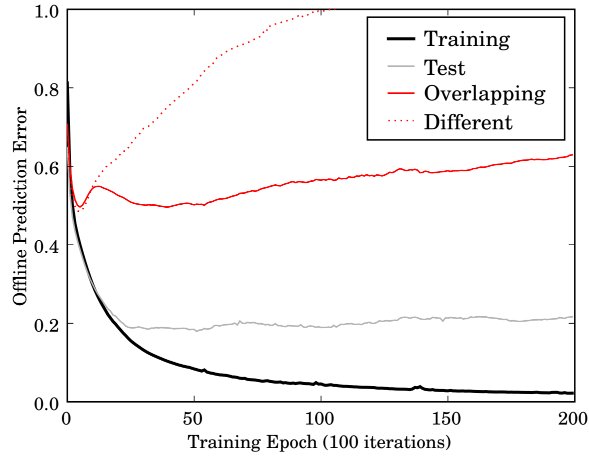


(b) 10 training examples

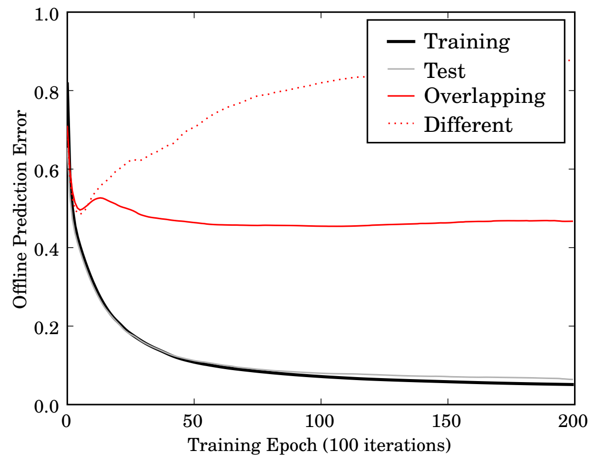


(c) 100 training examples

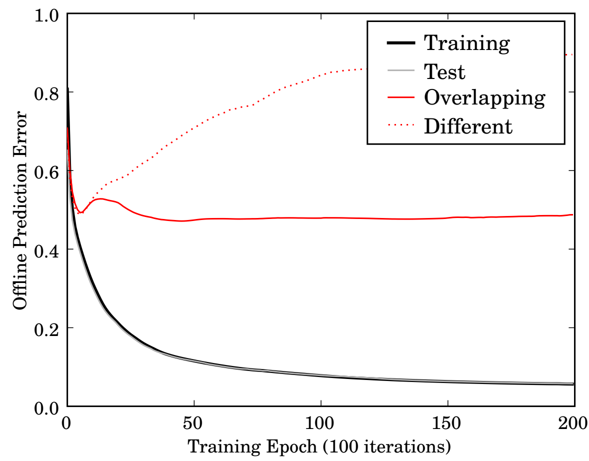
Figure 5.7: Training 10-step prediction networks with 20 hidden nodes and 1, 10, or 100 training examples. Results averaged over 30 experiments.



(a) 1 training example

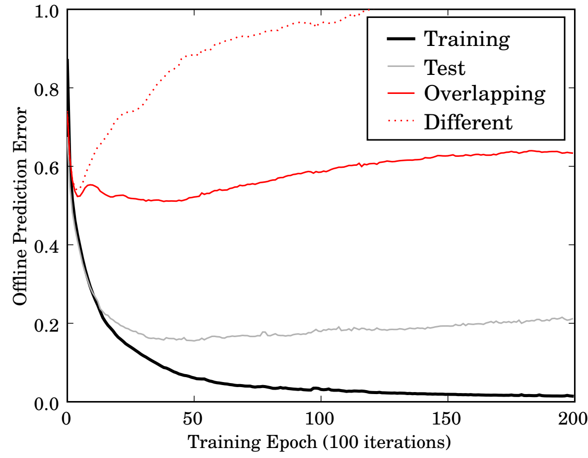


(b) 10 training examples

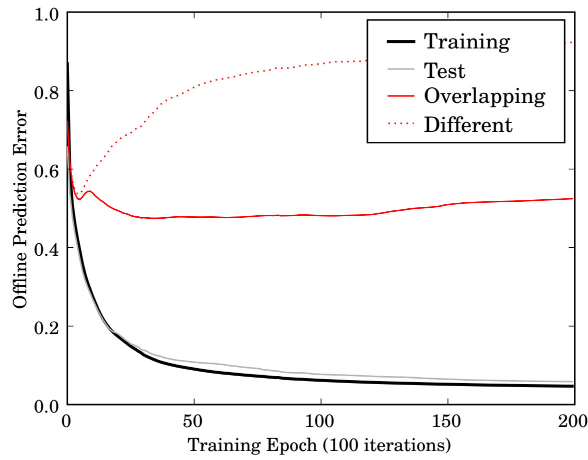


(c) 100 training examples

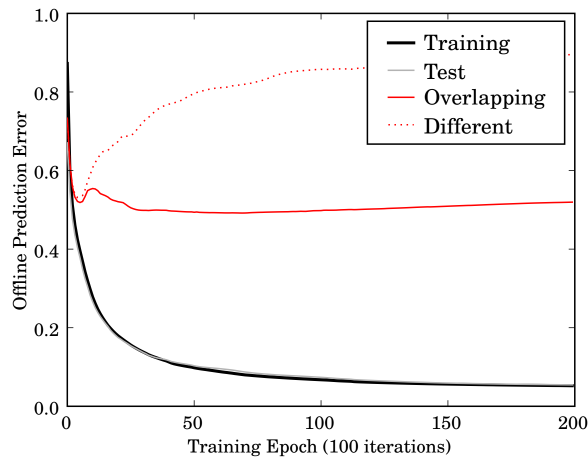
Figure 5.8: Training 10-step prediction networks with 40 hidden nodes and 1, 10, or 100 training examples. Results averaged over 30 experiments.



(a) 1 training example



(b) 10 training examples



(c) 100 training examples

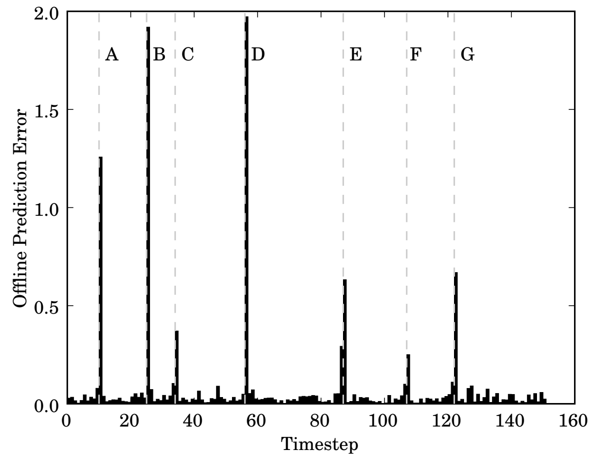
Figure 5.9: Training 10-step prediction networks with 80 hidden nodes and 1, 10, or 100 training examples. Results averaged over 30 experiments.

We hypothesized that the 1-step prediction networks would be good at predicting changes in sensor values, but bad at predicting changes in motor commands. This can be verified by looking at the prediction error per timestep from a trained anticipation network as the robot drives down the training path. Figure 5.10 shows the offline prediction errors per timestep for 1-step and 10-step prediction networks.

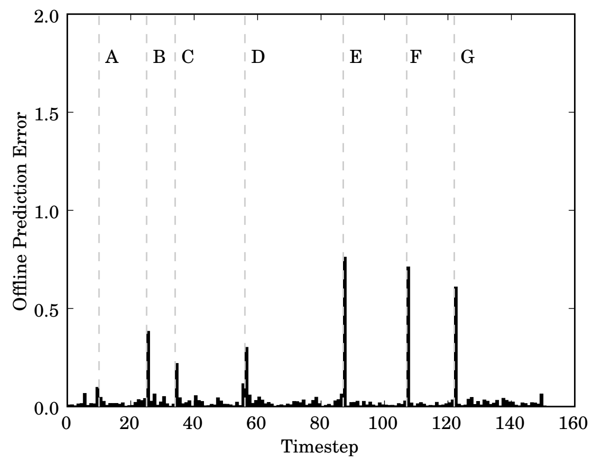
Figure 5.10(b) shows the 1-step offline prediction error from the 10-step prediction network. When we talk about the prediction error from an n-step prediction network we are discussing the n-step prediction unless otherwise noted. We show the 1-step offline prediction error here only because it makes a direct comparison between 1-step and 10-step prediction networks possible.

Figure 5.10(c) is rendered in gray scale. The total height of a bar indicates the offline prediction error at a timestep. In this case, the offline prediction error is an average of 10 separate prediction errors, and the color of the bar indicates how much error each prediction is contributing. Black indicates error contributed by short-term predictions, while light gray indicates error from long-term predictions. These figures are taken from a single experiment, but they are highly typical of 1 and 10-step prediction networks.

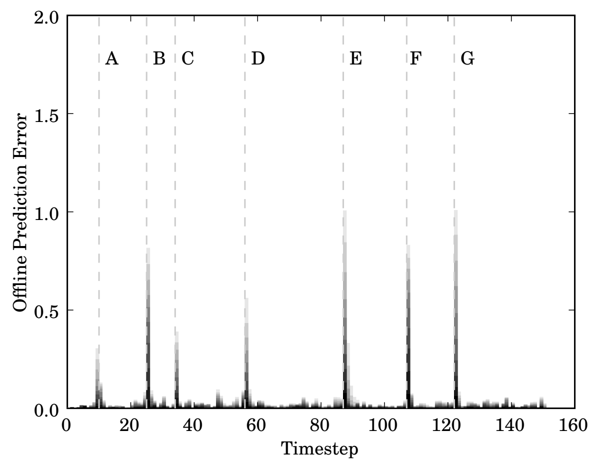
In Figure 5.10 we see that there are spikes in error from both prediction networks when the robot begins moving in a different direction. The spikes from the one-step prediction network reach nearly to 2, but the spikes from the 10-step prediction network reach only to 1. If the robot was travelling left (motor command: 0, 0, 1, 0) and turns upward (motor command: 1, 0, 0, 0), then the offline prediction error tells approximately how well the robot predicted the change. If the change was not predicted at all, the error will be the sum-squared difference between the commands.



(a) 1-step prediction



(b) 1-step prediction from 10-step network



(c) 10-step prediction

Figure 5.10: Offline prediction error from a 40-hidden node network trained with 10 training examples. Labeled timesteps correspond to events in Figures 5.1 and 5.2.

In this case the prediction error would be 2. If the robot was not sure which direction it should go (motor command: 0.5, 0, 0.5, 0), then the prediction error would be 0.5. Although the 10-step prediction network did not perfectly predict the motor command changes, it did see them coming.

The inability of the 1-step prediction networks to predict changing motor commands also explains why error was low on the overlapping and different paths through the grid world. Since the network primarily predicted that the current motor command should be maintained, prediction error is low on all grid world paths that have few turns and many long straight segments. 1-step prediction networks encoded little path specific information. In contrast 10-step prediction networks learned the defining characteristic of the path: the turns.

It is also worth noting that the 1-step predictions from the 10-step prediction network (Figure 5.10(b)) are more accurate than the predictions from the 1-step prediction network (Figure 5.10(a)). As we had hoped, the presence of long term predictions led to better short term predictions.

This section has shown that an anticipation network can be trained to predict future sensorimotor states. While the one-step prediction networks sometimes entirely missed the transitions between motor commands, the 10-step prediction networks achieved a reasonable ability to predict both motor commands and sensor readings.

5.2.2 Localization along a Path

In the previous section we measured performance based on the offline prediction error (Figure 4.10), but in this section we use the online prediction error (Figure 4.11).

Online prediction error is the error between successive approximations. It takes into account the most recent information instead of relying on predictions made long ago.

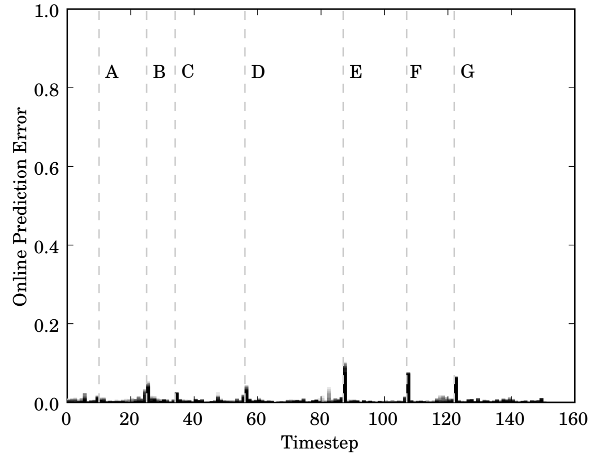
All tests in this section use one of the 10-step 40-hidden node prediction networks trained in the previous section with 10 training examples. The network was chosen at random. All other 10-step prediction networks should have similar properties to the one tested here.

First, we record the online prediction errors during one traversal of the original path (Figure 5.1), the overlapping path (Figure 5.2), and the different path (Figure 5.3). The results are shown in Figure 5.11.

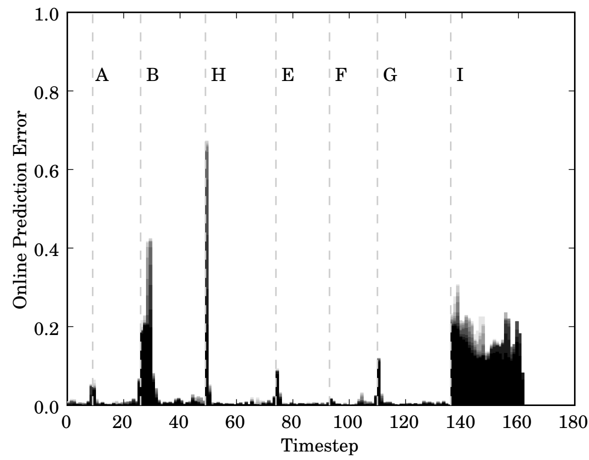
The fact that the online prediction error remains low when the robot drives down the original path indicates that the network has learned path-specific information. The robot does not know anything about other paths, so the online prediction error is significantly higher on the other paths in Figure 5.11. On the original path, the highest error is around 0.1. On the different path the error nearly reaches 1.0. Error is lower on the overlapping path, but error spikes when the robot leaves the known path. If nothing else, such a system could be used to determine when a robot leaves a known path.

In order to determine the extent of the localization occurring in the system, we can visualize selected hidden layer activations. Figures 5.12, 5.13, and 5.14 show the activations of three hidden nodes as the robot traverses six different paths.

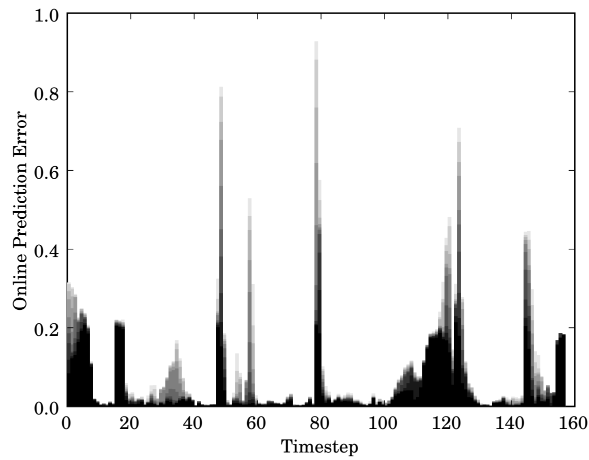
In Figure 5.12 we see that the 17th hidden node always activates and deactivates in approximately the same places. Particularly, the node always activates about the same distance before the last turn. As shown in Figures 5.13 and 5.14, nodes 22 and 28 have even more specific activation patterns. Both of them activate for only a



(a) Original path



(b) Overlapping path



(c) Different path

Figure 5.11: Online prediction errors from a 10-step 40-hidden node prediction network as the robot drives down the original path, the overlapping path, and the different path. Labeled timesteps in Figure 5.11(b) correspond to events in Figure 5.2.

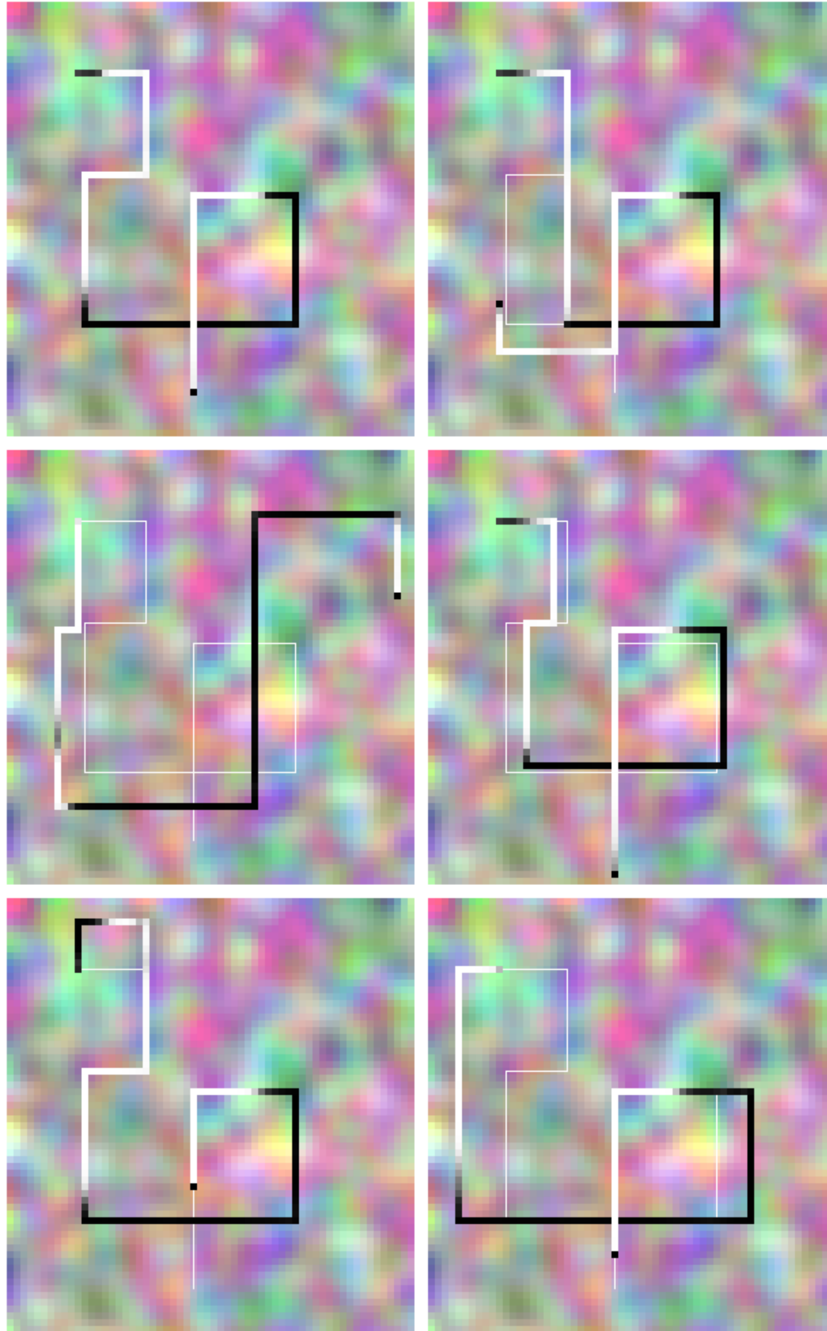


Figure 5.12: Activation of the 17th hidden node in a 10-step 40-hidden node prediction network as the robot drives down six paths through the environment. The original path is at the top-left, and the path from Figure 5.2 is at the top right. The node activates early in the run and again in the last quarter of the run.

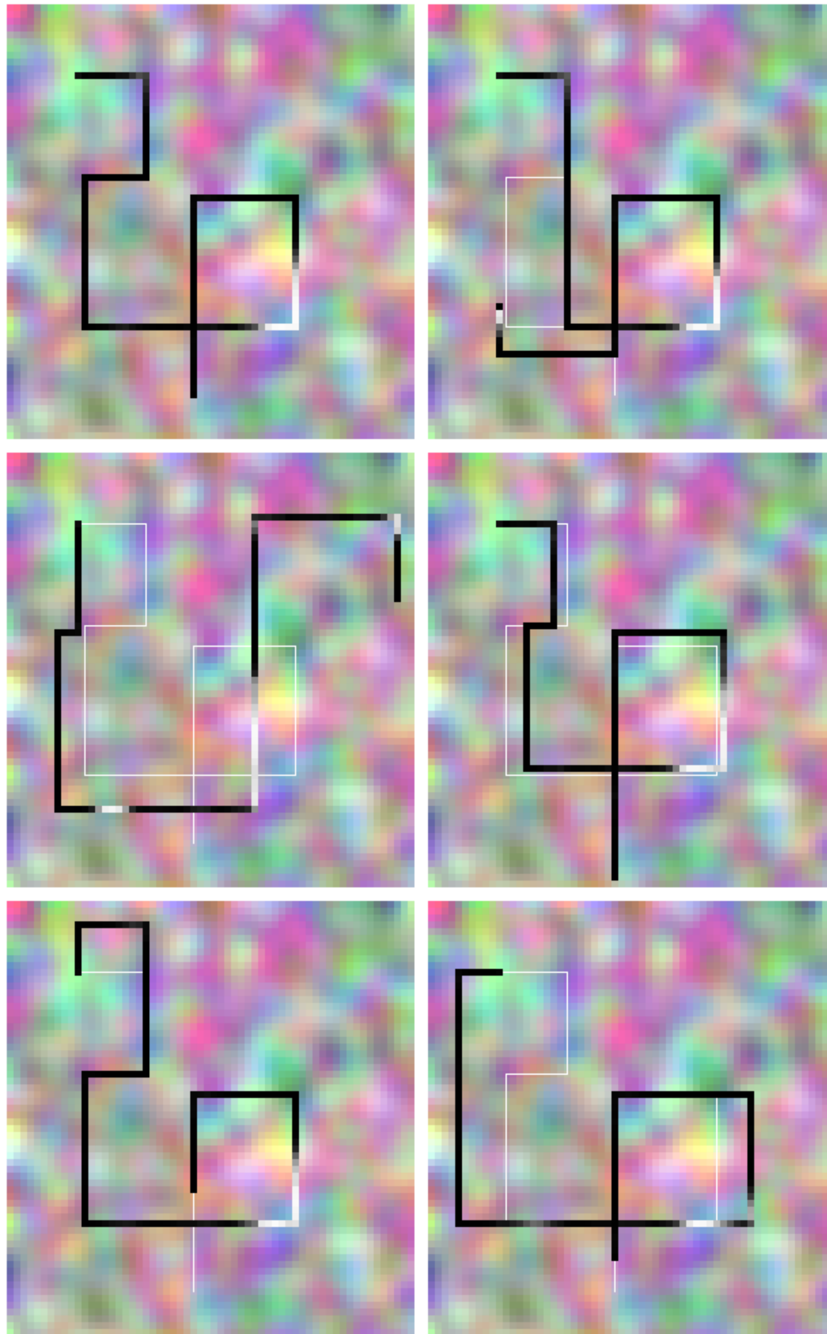


Figure 5.13: Activation of the 22nd hidden node in a 10-step 40-hidden node prediction network as the robot drives down six paths through the environment. The node usually activates at the position labeled D in Figure 5.1.

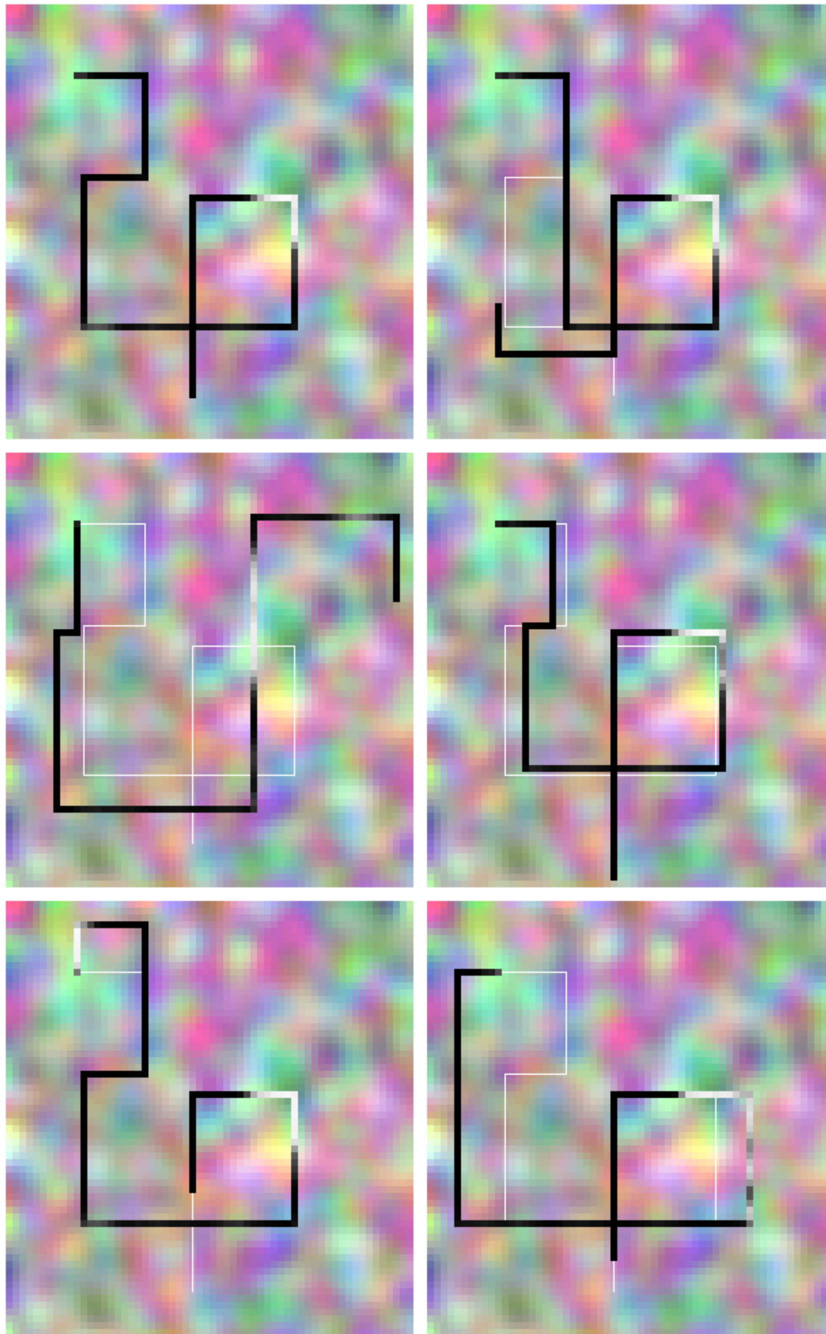


Figure 5.14: Activation of the 28th hidden node in a 10-step 40-hidden node prediction network as the robot drives down six paths through the environment. The node usually activates at the position labeled *E* in Figure 5.1.

single corner. The three hidden nodes activations shown here had the most obvious patterns. It is likely that other positions on the path could be described by linear combinations of hidden node activations. The presence of neurons that only activate at certain positions shows that the network is able to localize itself along the trained path.

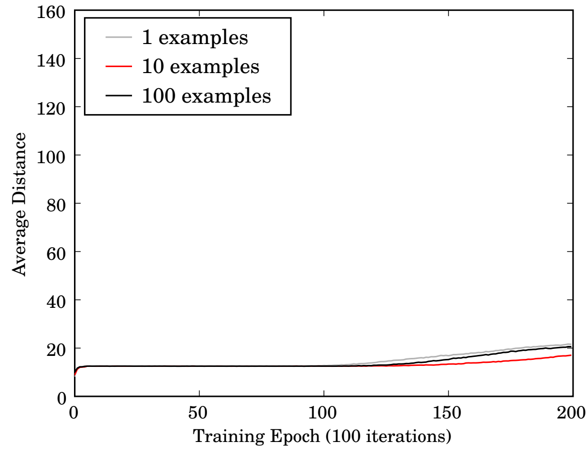
5.2.3 Control from Prediction

In the previous section the robot demonstrated that it could maintain a concept of its current location as it moved along a path. Knowing where you are is not extremely useful if the information can not also be used for online robot control. In this set of experiments we attempt to use the learned networks to control the robot.

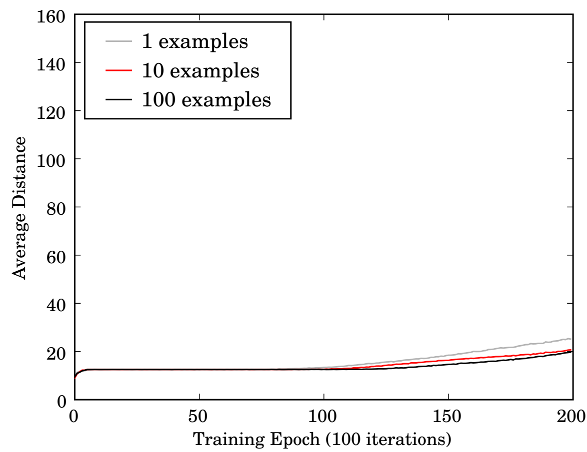
We use the predictive control algorithm from Figure 4.9. The predictive control algorithm uses a variable number of predictions depending on the value given for the look ahead length, here we will use only the last prediction. In order to set the motor command of the robot, we select the action from Table 5.1 that matches the predicted command as closely as possible.

While the networks were being trained in Section 5.2.1, we also measured the robustness of predictive control. After every 100 training iterations of each prediction network, we attempted to control the robot with the predictive control algorithm. Given 100 predictive control attempts, we measured the average distance traveled along the path and counted the number of successful completions of the path. As long as a robot stayed within one grid cell of the true path it was allowed to continue.

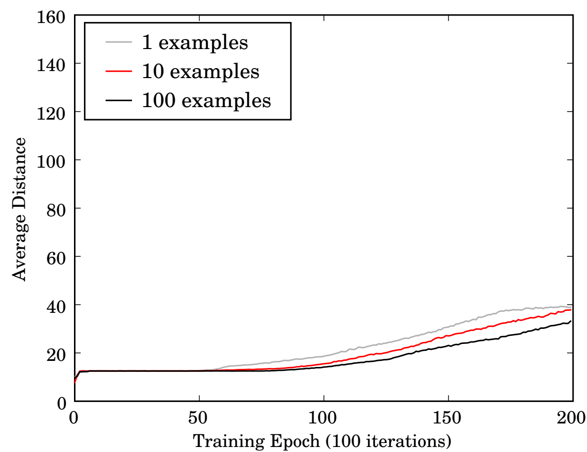
Figure 5.15 shows the performance of the 1-step prediction networks. The 80 hidden node network performs the best, but it only makes it about one fourth of the way down the path. No attempts at predictive control with a 1-step prediction network reached the end of the path. The inability of 1-step prediction networks to predict changes in motor commands makes them poorly suited for this task. Figure 5.16 shows the average distance traveled under predictive control and the percentage of attempts that followed the path all the way to the end.



(a) 20 hidden nodes

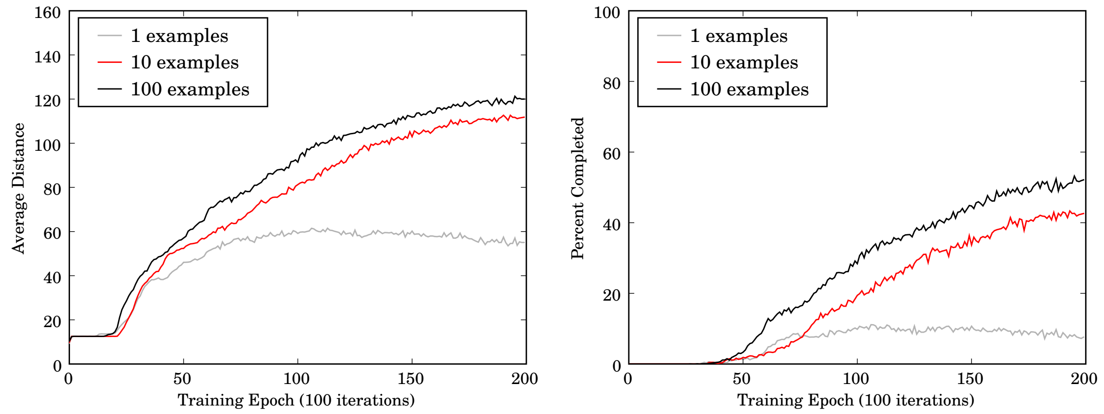


(b) 40 hidden nodes

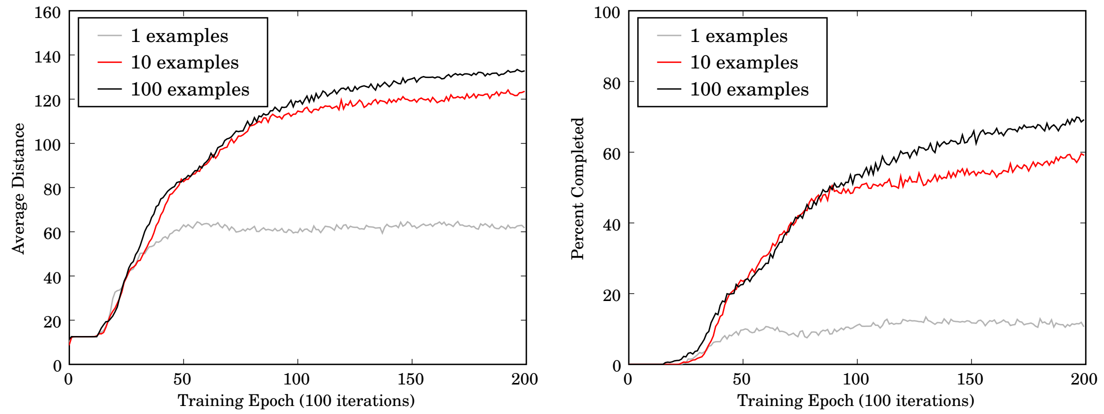


(c) 80 hidden nodes

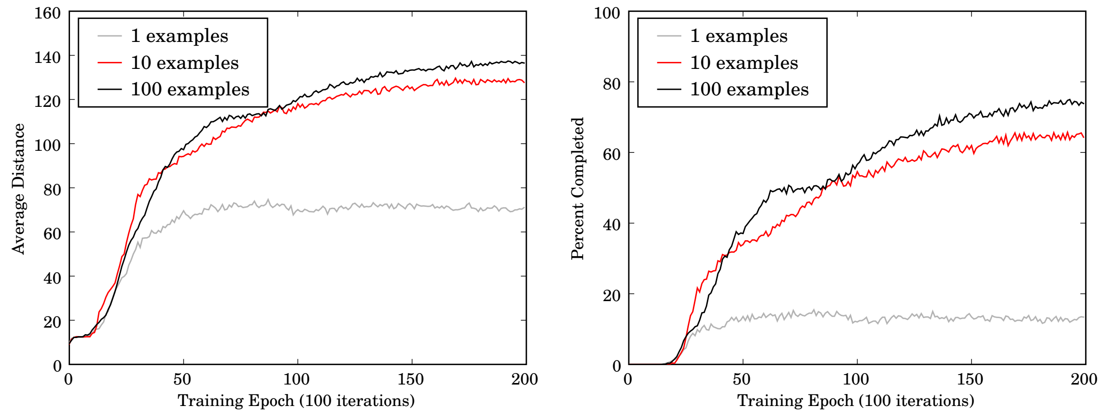
Figure 5.15: Average distance robot travels before falling off the path when being controlled with the predictive control algorithm and a 1-step prediction network. Results averaged over 30 experiments.



(a) 20 hidden nodes



(b) 40 hidden nodes



(c) 80 hidden nodes

Figure 5.16: Performance under predictive control with a 10-step prediction network. *Left:* Average distance robot travels before falling off the path. *Right:* Percentage of attempts completed. Results averaged over 30 experiments.

For each number of training examples, the 10-step 80-hidden node prediction networks achieved the best performance. With 100 examples, the network could achieve over 75% accuracy. Accuracy fell off as the number of training examples was reduced, but the change from 100 training examples to 10 training examples did not have a large effect. It is likely that using more than 100 training examples would have little effect on performance.

The most interesting aspect of these results is the performance of multi-step prediction networks trained with only a single training example. Given the amount of overfitting shown in Figures 5.7, 5.8, and 5.9, it would seem that a network trained with only a single training example would be of little value. However, we have shown that such a network is more successful at following the path than any of the 1-step prediction networks. The 1-step prediction networks had substantially lower offline prediction errors on the test set, yet none of them were able to complete the path.

In this section we have seen the power of predictive control. Starting from a single training example, it was possible to duplicate the behavior. In the next section we will apply reinforcement learning to attempt to learn more robust control policies.

5.2.4 Reinforcement Learning

In this section we demonstrate an alternate use for the anticipation network we have learned. An anticipation network generates a continuous stream of predictions. As we have seen in the previous section, when we are on the right path the prediction error will be low, and when we are not on the right path it will be high. The prediction error can therefore be used as a form of constant feedback. Using predictive

control we could only follow the path 75% of the time. It is hoped that RL can be used to create more robust policies.

Our reinforcement learning system uses prediction error as feedback in order to learn how to follow the path. In the grid world task, we can use the current sensorimotor state and the hidden layer of the prediction network as a state representation. The possible actions are left, right, up, and down. The only remaining choice is the reward structure.

Recurrent networks can behave unpredictably, so we do not want to directly estimate the prediction error from the network. Instead, we will apply a threshold. When the online prediction error is below 0.05, we will give a reward of one. When the online prediction error is above 0.2, we will give a penalty of one and terminate the episode. When the error is between 0.05 and 0.2, no reward or penalty will be given. These values were chosen such that if the path is followed perfectly, reward will be received on nearly every timestep. Refer to Figure 5.11 for the online prediction errors along the path.

Q-Learning (see Section 2.2.2) was used to learn a control policy. Ideally, this policy should follow the training path. The learning rate was set to 0.01 and the discount rate was set to 0.9. These are reasonable parameter values for Q-Learning. The discount rate is somewhat low, but this is acceptable because rewards are received constantly during each episode. We do not need to look ahead a long time to determine if an action is good. A neural network with a 30 node hidden layer was used to approximate the action-value function, and momentum of 0.5 was used throughout training.

In order to encourage exploration at the proper times, the probability of taking a random action, ε, was varied. As training progressed, a counter was kept that indicated the farthest any episode had successfully followed the path. If we were at the start of an episode, ε was set to a low value in order to capitalize on the knowledge already acquired. As we approached the longest recorded episode length, ε was adjusted higher in order to encourage exploration of the less well known parts of the path. The total range of ε was from 0.0001 to 0.2, but intermediate values were used most often. This was necessary because the reward structure was set up so that the episode would usually end when the robot fell off the path. If ε is too high, the robot is never able to make it far into the episode because random actions knock it off the path and end the episode.

The robot is given 180 timesteps to complete the path, which is more than enough time. No reward is given at the end of the path, because we want to learn with only internal information. Also, the episode is not ended when the robot leaves or reaches the end of the path.

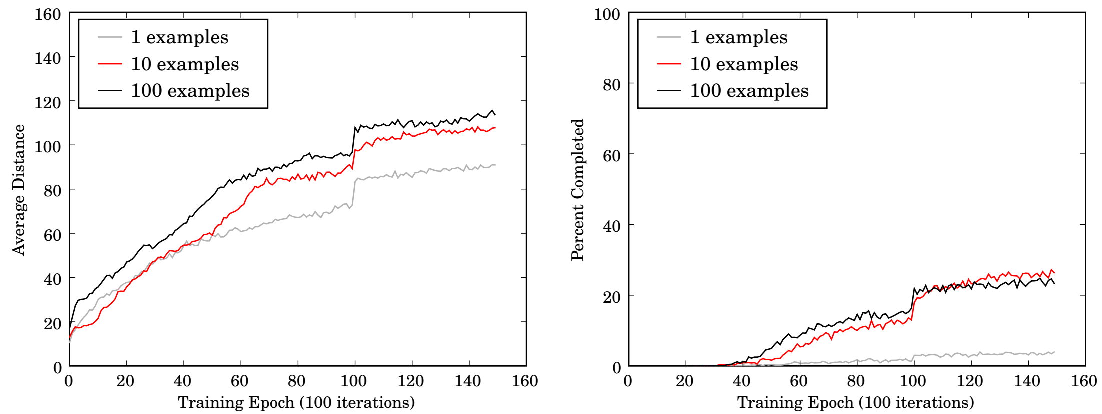
Tests were conducted on each 10-step prediction network trained in Section 5.2.1. Each agent was trained for 10,000 episodes. After 10,000 episodes, ε was set to zero and the agent was allowed to run for another 5,000 episodes in order to evaluate performance of the greedy policy. The results are presented in Figure 5.17. The left side of Figure 5.17 shows the average distance traveled in each epoch, and the right side shows the percentage of attempts that stayed within one grid cell of the path all the way to the end. Training was not attempted with 1-step prediction networks.

Note that the distances in Figure 5.17 are the average distance travelled and not the average distance along the path. It is possible that an agent could stay alive

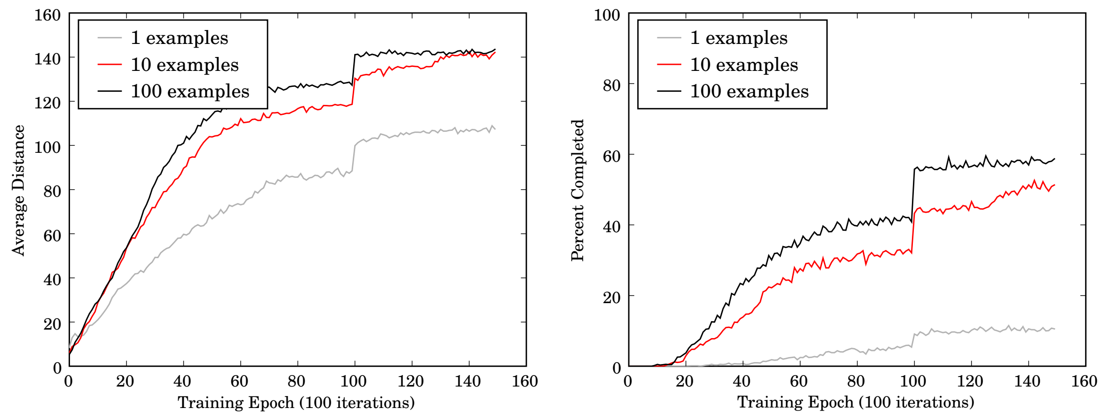
and receive reward without following the intended path. Therefore, the completion percentages gives a better indication of the real performance of the system.

When training was completed, the agents reached performance similar to the predictive control algorithm in some cases. The agents based on the 80-hidden node prediction network performed comparably, but the performance of the agents using the 20-hidden node networks was substantially worse than under predictive control. While this is not the result we had hoped for, it demonstrates that it is possible to extract useful information from a multi-step prediction network.

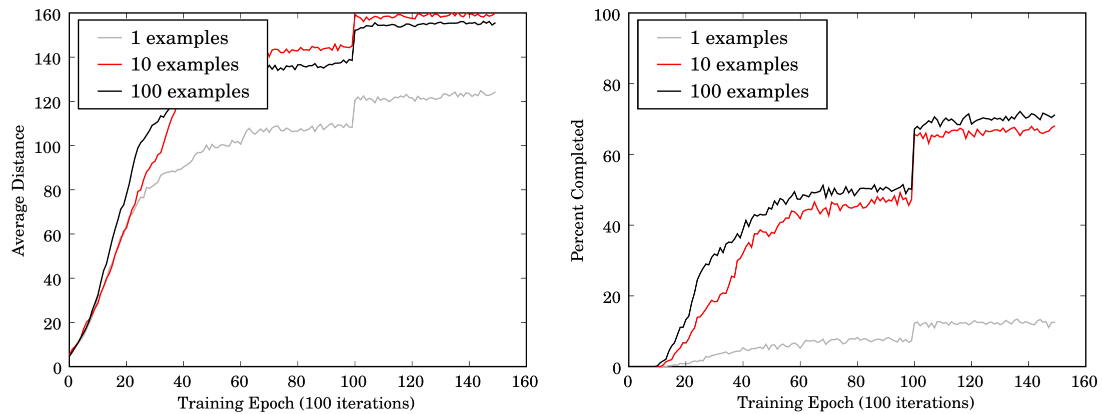
We have shown that an anticipation network can be used to generate a reward signal for reinforcement learning. It is hoped that with more advanced RL algorithms performance could be substantially increased.



(a) 20 hidden nodes



(b) 40 hidden nodes



(c) 80 hidden nodes

Figure 5.17: Performance of Q-Learning agents based on 10-step prediction networks. *Left:* Average distance robot travels during an episode. *Right:* Percentage of attempts completed. At epoch 100 ε is set to zero. Results averaged over 30 experiments.

Figure 5.18: The simulated robot domain and the path for the robot to follow. The robot starts at the upper left. Each grid square is one meter long.

5.3 Simulated Robot

The grid world used in the previous section had one important feature common to real sensor data: sensor readings are not independent events. It is possible to accurately predict sensor readings by looking back at past readings and building a model of the environment. This property allows most of the what was done in the grid world to be transferred to a more realistic environment without change. The tests in this section are not as thorough as the grid world tests, but they will verify that our approach works in a realistic environment.

To simulate a realistic environment for a robot, we used the Stage simulator (Gerkey et al., 2003). The packaged "hospital" environment was used. This environment is based on the US Army hospital on the grounds of Fort Sam Houston in San Antonio, Texas. This combination of simulator and environment has been used extensively in robotics research, and is characterized by smooth code transition to

	Action	Representation
1.	Stop	(1, 0, 0, 0, 0, 0)
2.	Move Forward	(0, 1, 0, 0, 0, 0)
3.	Turn Left	(0, 0, 1, 0, 0, 0)
4.	Turn Right	(0, 0, 0, 1, 0, 0)
5.	Forward Left	(0, 0, 0, 0, 1, 0)
6.	Forward Right	(0, 0, 0, 0, 0, 1)

Table 5.2: All possible actions in the simulated hospital.

real robots. A screenshot of the simulator and path is shown in Figure 5.18. The path is about 20 meters long, and it takes the robot about 47 seconds to finish the path.

We use a Pioneer2AT like robot with a ring of 16 sonar sensors. No other sensors are used. The 16 sonar sensors can detect obstacles up to 5 meters away. This configuration should be adequate for implicit localization. The Pioneer2AT is controlled by specifying a straight-line velocity and a rotational velocity. The motor state consists of these two values plus 6 other values that specify the current command the robot is executing. Possible commands are listed in Table 5.2. On the training path, the robot never used the stop command or turned without driving forward. Commands are sent every 100ms.

In the grid world we used a variable number of training examples, but in the simulated hospital we will only use one. While it was straight-forward to collect multiple training examples in the grid world, it is more difficult in Stage. A person could drive the path multiple times, or we could use an algorithm to retrace a path exactly. If we choose the first option the paths driven by a person may differ too much and make learning difficult. If we use an existing algorithm to follow the path,

then it is unclear that we are gaining anything by teaching another system to do a task that was already possible. Neither of these situations are ideal.

We demonstrated that is possible to learn to repeat a task based on only a single example (see Figure 5.17). The controller trained with only one training example had a much lower completion percentage than the others. However, it did manage to reach the end of the path on a sizeable percentage of the attempts. We hypothesize that we will be able to achieve similar performance in the simulated hospital.

5.3.1 Abstraction

Unlike the grid world example, the sensor data from the Stage simulator can benefit from abstraction. Because of the sonar configuration and environment structure, certain combinations of sonar readings are likely, while others are unlikely or impossible. Using abstraction will eliminate the need for the anticipation layer to interpret individuals readings, and will simplify the anticipation network structurally by requiring fewer neurons for the more compact representation.

The abstraction network is trained to compress 16 sonar readings into an abstract representation of 8 values. The network has 16 inputs, 16 outputs, and an 8 node hidden layer. The hidden layer uses a sigmoid activation function while the output layer uses a linear activation function. The network is trained for 10,000 passes through the sonar data from the single training example. The learning rate is set to 0.0001 and momentum of 0.5 is used. The result of training is shown in Figure 5.19.

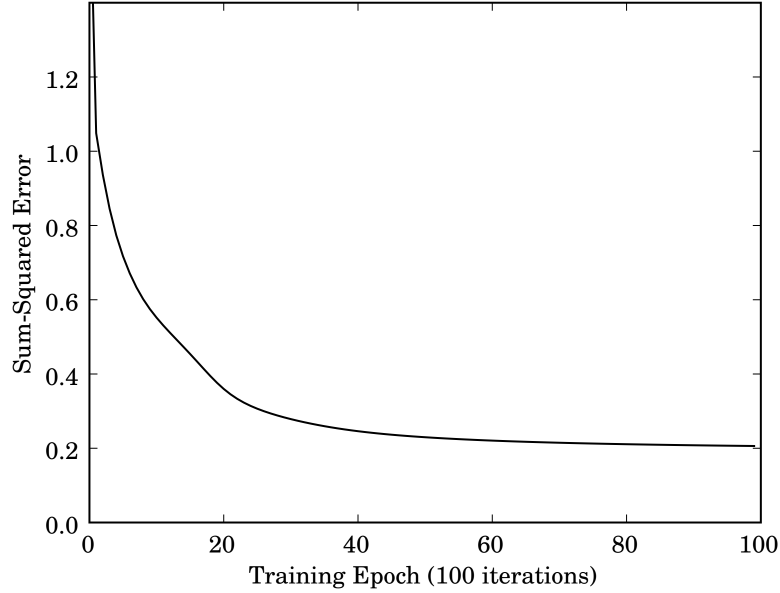


Figure 5.19: Sum-squared error during training of an sonar abstraction network. The network takes 16 sonar values, and compresses the information into 8 continuous values.

5.3.2 Anticipation

The anticipation layer is very similar to that used in the grid world. The input into the anticipation layer has 8 values for the abstract sonar representation, 2 values for the motor velocities, and 6 values for the actions. Therefore the network needs 16 inputs and 16 outputs for each prediction.

Although 1-step prediction was not adequate for the grid world, and we do not expect it to be here, examining what can be learned by a 1-step prediction network with more realistic data gives us a better picture of the limits of such a network. In order to force the network to learn the transitions, we will also use an 80-step prediction network. A 10-step prediction, as was used for the grid world, would only predict outputs for the next second. However, we are attempting to predict a longer sequence here so we increase the prediction length to compensate. When

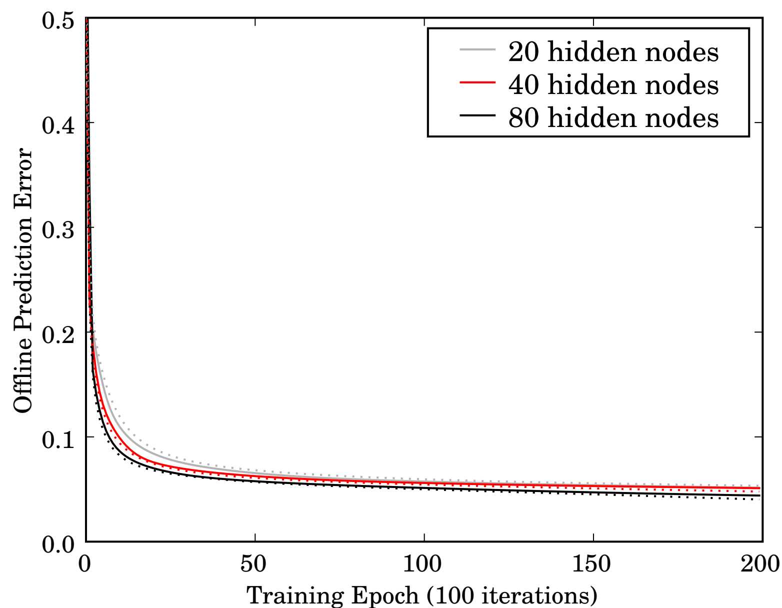


Figure 5.20: Offline prediction error during the training of a 1-step prediction network. The solid lines each represent one trial, while the dotted lines are the average result from 10 experiments.

performing 80-step prediction, there are 16 outputs for each prediction and the combined network has 1280 outputs.

The experimental setup was the same as in the grid world. The learning rate was set to 0.001, the momentum was set to 0.5, and all nodes in the network used sigmoidal activation functions. Once again, anticipation networks with 20, 40, and 80 hidden nodes are evaluated. Figures 5.20 and 5.21 show the training progress for the prediction networks. Figures 5.22 and 5.23 show the offline prediction errors after training is completed.

While the offline prediction error is usually low, it jumps very high when the robot changes actions. In the one-step prediction networks this indicates a fatal flaw; they are incapable of predicting when actions change. However, when using an 80-step prediction network the width of the spikes is also important. When the spikes

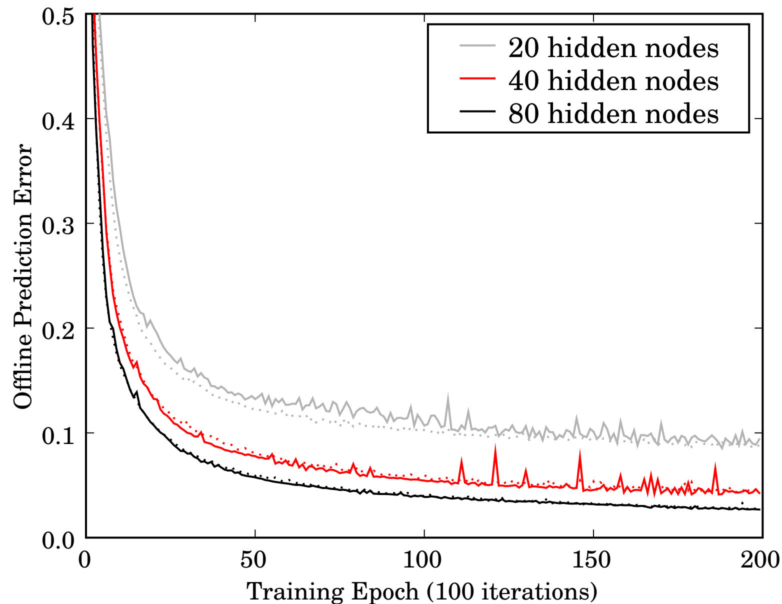
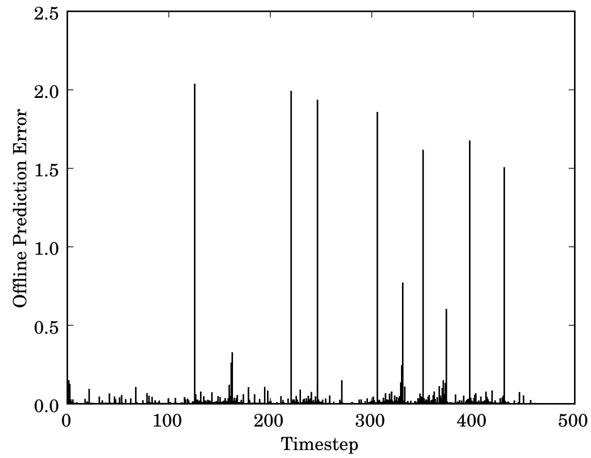


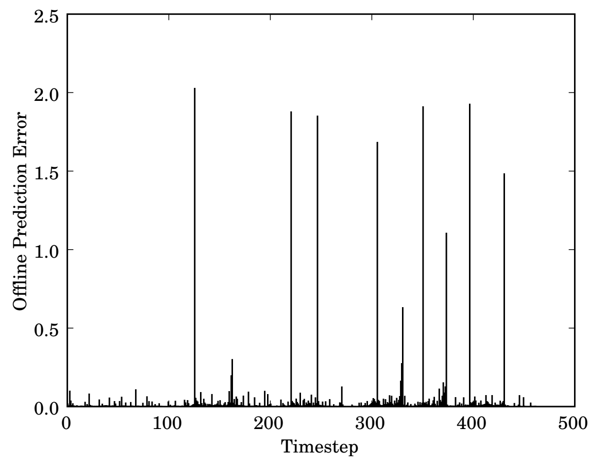
Figure 5.21: Offline prediction error during the training of a 80-step prediction network. The solid lines each represent one trial, while the dotted lines are the average result from 10 experiments.

are wide, as in Figure 5.23(a) at timestep 125 or Figure 5.23(b) at timestep 250, it means that long term predictions are slow to pick up the change. The 80-step 80-hidden node network has no wide spikes. In this case, all motor command changes are predicted, but some will be predicted up to a second after they should have been. While this will degrade performance, it should still be possible to complete some runs.

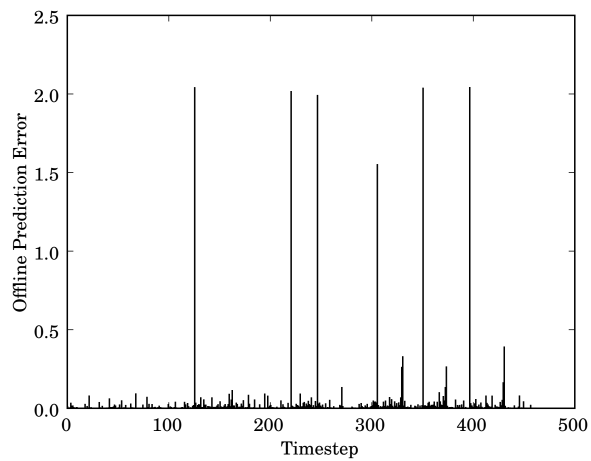
Given the large number of outputs, it is somewhat surprising that the 80-step prediction networks are able to learn to output accurate predictions with only 20, 40, or 80 hidden nodes. The 80-step 80-hidden node prediction network outperformed the 20 and 40 hidden layer networks. The 80-step 80-hidden node network will be used for the remainder of this section.



(a) 20 hidden nodes

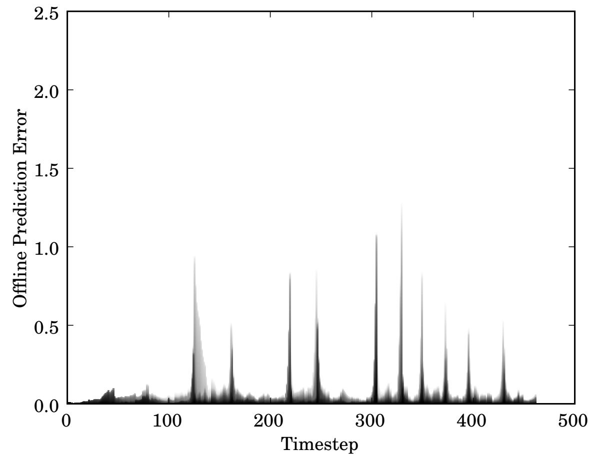


(b) 40 hidden nodes

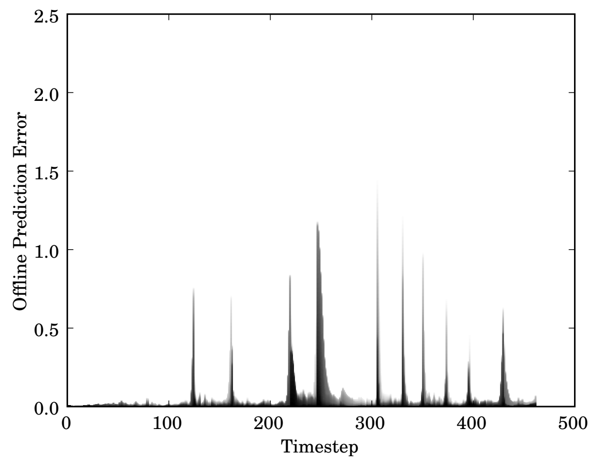


(c) 80 hidden nodes

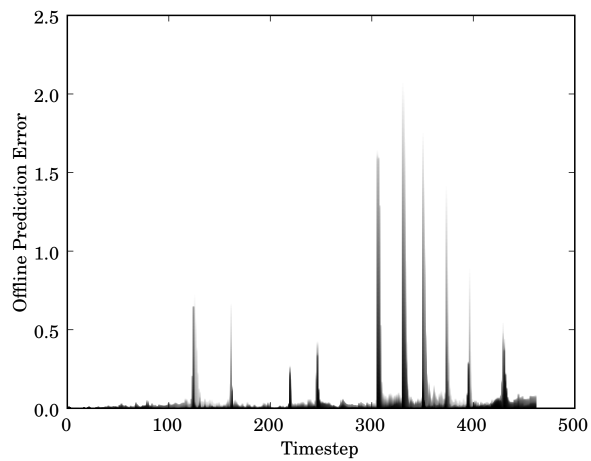
Figure 5.22: Offline prediction errors from 1-step prediction networks as the robot follows the path.



(a) 20 hidden nodes



(b) 40 hidden nodes



(c) 80 hidden nodes

Figure 5.23: Offline prediction errors from 80-step prediction networks as the robot follows the path.

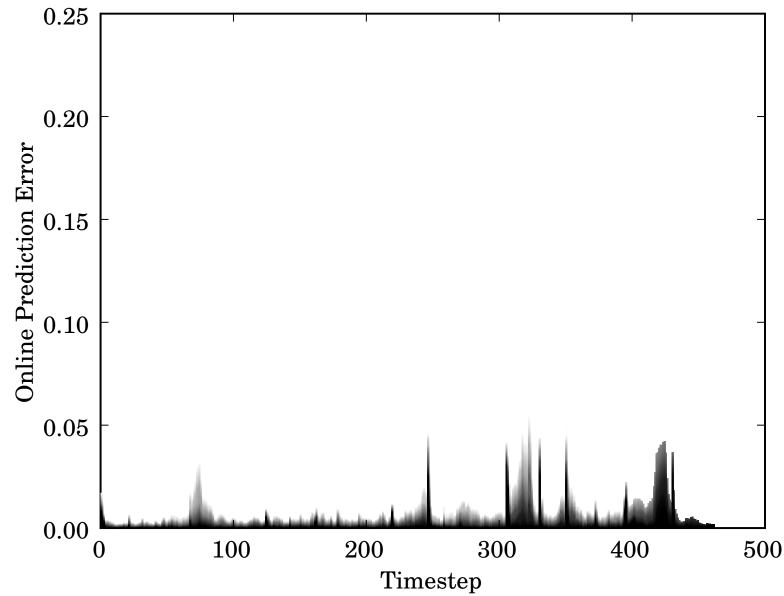


Figure 5.24: The online prediction error from the 80-step 80-hidden node prediction network as the robot follows the path.

5.3.3 Localization along a Path

When examining the grid world results, we attempted to determine if the prediction network was capable of global localization. With such an abstract problem domain, it was difficult to determine exactly what was happening. Using a simulated robot with sonar sensors puts us in a much more comfortable place to answer this question.

Figure 5.24 shows the online prediction error on the path that the robot was trained on. As expected the error stays extremely low. There are several spikes in the online prediction error, but all errors are low enough that the network can accurately predict the future as long as it stays on the path.

Figure 5.26 shows the online prediction error on as the robot drives down the completely new path shown in Figure 5.25. Figure 5.28 shows the error on the path in Figure 5.27. Because the path in Figure 5.27 overlaps with the initial training path we expect it to have a much lower error than the completely new path. Both

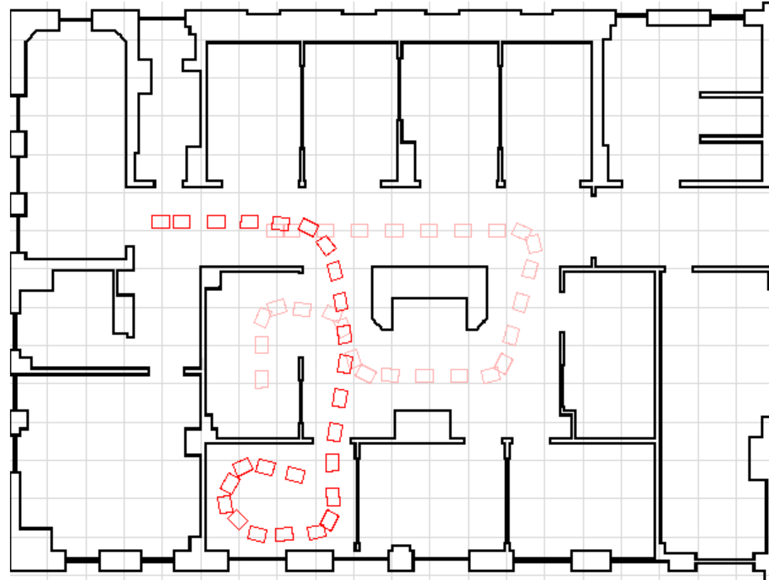


Figure 5.25: A different path for the simulated robot. The original path is shown in a lighter color than the new path.

the original and overlapping paths were driven by a person, so the correspondence between them is far from exact. Still, the online prediction error on the overlapping path is noticeably lower than the error on the different path. The total error on the overlapping path is 19.88, while the total error on the new path is 36.20.

These graphs demonstrate that an anticipation network is able to make a distinction between familiar and unfamiliar states in a realistic navigation problem, even when the paths do not completely overlap. This shows the power of the network to perform implicit localization. The robot is capable of recognizing sequences of states it has seen before and using that information to accurately predict what will come next.

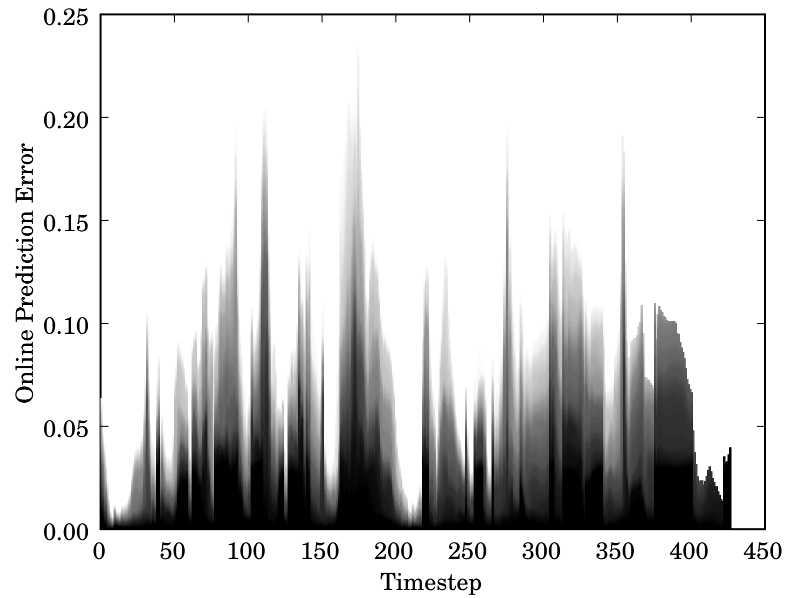


Figure 5.26: Online prediction errors from the 80 node hidden layer, 80-step prediction network as the robot drives down the path in Figure 5.25.

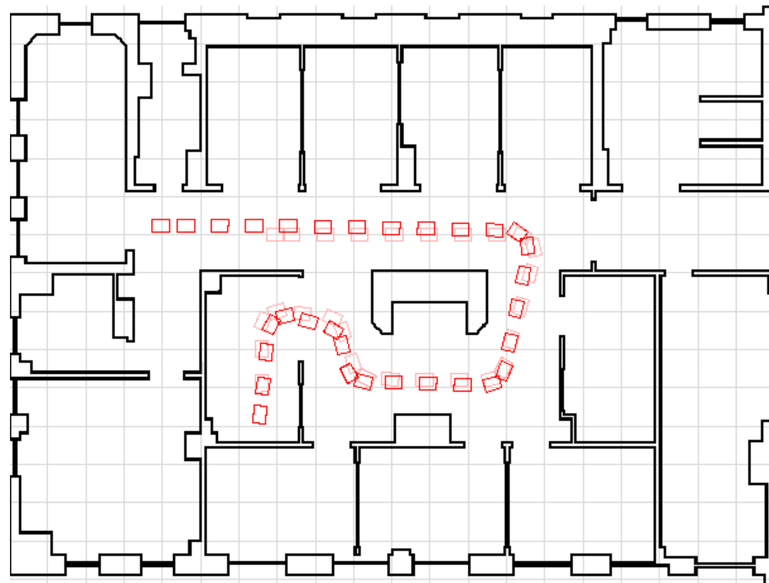


Figure 5.27: A similar path for the simulated robot. The original path is shown in a lighter color than the new path.

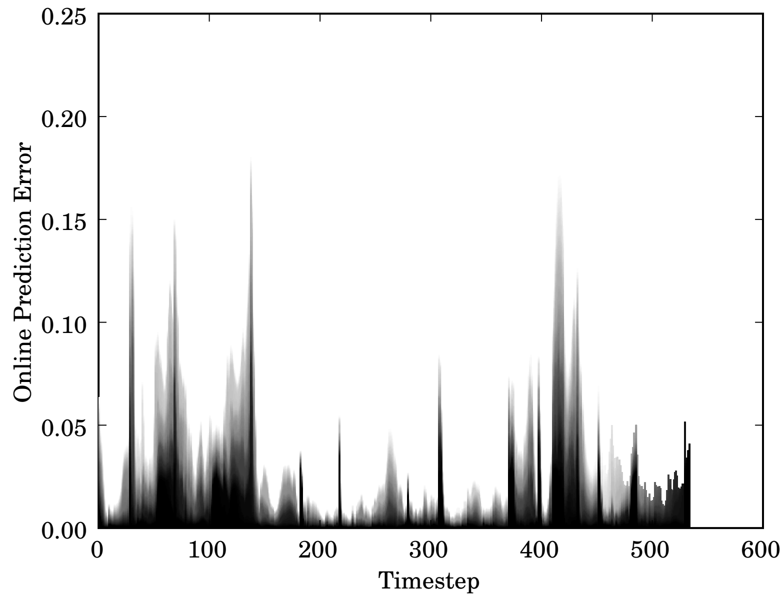


Figure 5.28: Online prediction errors from the 80 node hidden layer, 80-step prediction network as the robot drives down the path in Figure 5.27.

5.3.4 Control from Prediction

It has been shown that prediction networks are capable of recognizing parts of the paths that are similar to the initial training data. In a very loose sense, some form of localization must be occurring, but it is much more impressive if we can see a robot intelligently control itself. In this section we will examine robot control based on predictions.

It is worth noting a difference between memorization and understanding. Memorization is not an intelligent process, it does not require any knowledge about the information it holds. Understanding, on the other hand, demonstrates knowledge of the data. If the prediction network is merely memorizing a sequence of motor commands, it is not doing anything that could not be done easily with a sensorimotor log stored in a flat file.

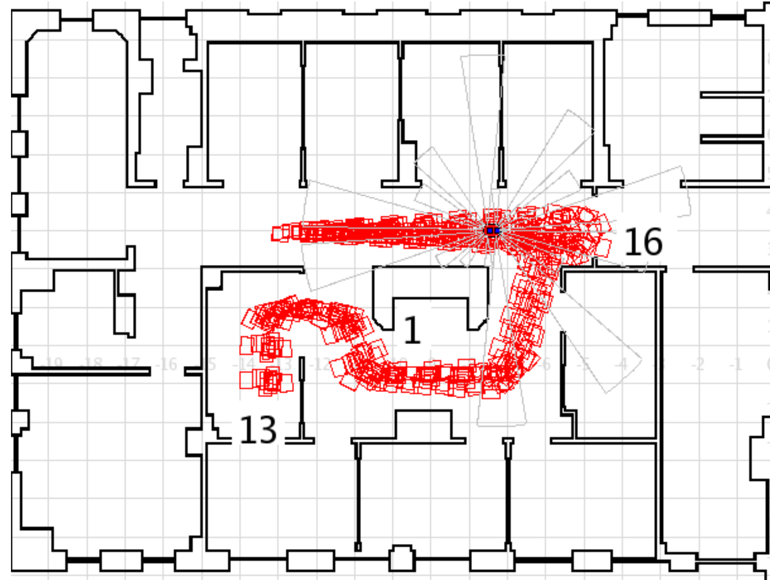


Figure 5.29: Attempting to repeat a path by exactly duplicating the motor commands used on the original path. The initial position for each trial is randomly offset from the initial start position. The robot is rotated ±4 degrees, is moved up to 2 meters forward or 0.5 meters back, and is offset laterally by up to 0.1 meters. Numbers on the image mark the general area where the robot collided with walls.

Figure 5.29 demonstrates the results of memorizing motor commands on 30 attempts to follow the path. In each attempt the original motor commands are repeated, regardless of sensor input. This strategy would work in a completely deterministic environment. However, if the environment is not deterministic, then playing back recorded motor commands will not be able to accomplish the original task. In order to demonstrate this, the starting position is randomly changed. When the starting position is moved sufficiently, the robot will drive straight into walls.

Figure 5.30 shows the robot controlling its motors using the 80-step 80-hidden node anticipation network developed previously and the predictive control algorithm (see Figure 4.9) with a lookahead length of 80. In order to set the motor command of the robot, we select the action from Table 5.1 that matches the predicted command

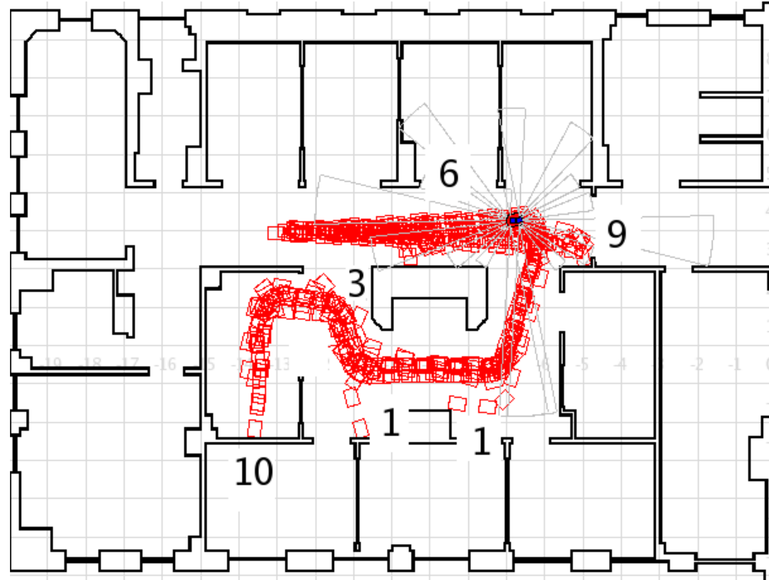


Figure 5.30: Attempting to repeat a path by using the predictions from an anticipation network. The starting positions are distributed as in Figure 5.29. Numbers on the image mark the general area where the robot collided with walls and how many times such collisions occurred.

as closely as possible. The same starting positions were used for Figure 5.29 and Figure 5.30.

The robot follows a much more recognizable path in Figure 5.30. After the first corner the robot following the memorized path was significantly misaligned with the training run. In contrast, the robot taking its movements from the predictive control algorithm was much better aligned. The prediction network was capable of choosing when to turn such that the robot would be aligned with the correct path.

Both strategies made it to the destination approximately the same number of times. The prediction network sometimes got badly confused at the corners, while the memorized motor commands led the first robot straight into walls. However,

when it did not get confused, the prediction network did a very good job of keeping the robot on the right path, even on trials in which the starting position was significantly offset.

5.3.5 Reinforcement Learning

Reinforcement learning is used in the Stage simulator in a similar way to how it was used in the grid world (see Section 5.2.4). However, an important difference is that the robot is not always started at the same position. Instead, the robot's starting position is varied as in Figure 5.29. In the grid world the robot could learn to follow the exact path. With a random starting distribution in Stage, it is now impossible for the robot to follow exactly the same path every time. Instead, the robot will have to learn a new path that is similar, but not completely the same as the original.

In the previous section we showed that a robot traversing a similar path had smaller online prediction errors than a robot traversing a completely different path. We will use RL to reward the agent for finding similar paths. In the grid world, we gave harsh penalties for even small errors. Here we will have to take a different approach, and give rewards even when errors are significantly higher than would be seen when following the original path. While we set the discount rate to a low value (0.75) in the grid world, we will need a significantly higher discount rate here. The actions that maximize immediate reward may not be the best actions to take in the long term.

We will give the agent a reward whenever the online prediction error is less than 0.1875. When the error is greater than 0.1875 the episode will be terminated. This value is high enough that the path in Figure 5.27 would be allowed, but if the

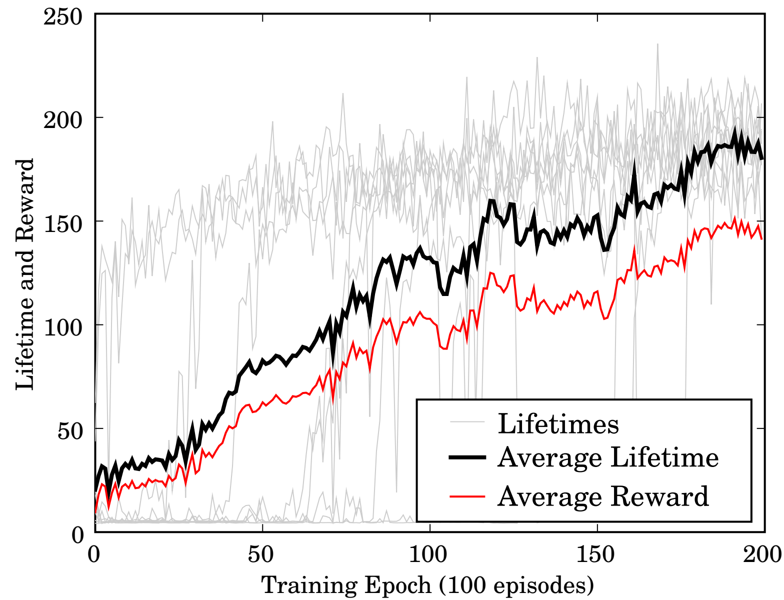


Figure 5.31: Using reinforcement learning to learn simulated robot control. Results of 10 runs of the Q-Learning algorithm.

agent tried to take the path in Figure 5.25 the episode would be terminated. When the online prediction error is less than 0.03, we will give the agent a reward of 1, otherwise the reward is always 0.1. An agent can only get the higher reward if it follows the original path very closely.

The discount rate was set to 0.96 and ε was set to 0.01. The learning rate was held constant at 0.01. Training ran for 20,000 episodes, and the results are shown in Figure 5.31. On average, the robot learned to follow the path for around 200 timesteps. The robot following the learned policy is shown in Figure 5.32.

The Q-Learning system displayed some of the same problems that the predictive control system had. Most notably, both systems often got lost at the first corner and went straight instead of turning. Out of 30 trials, the agent missed the first corner 12 times. In comparison the predictive control algorithm only did this nine times, and the memorized motor commands missed the turn 16 times.

Figure 5.32: Q-Learning controller attempting to drive the robot down the path after being trained for 20,000 episodes. The greedy policy is followed, and episodes are not terminated when the online prediction error exceeds the limit set for training. The starting positions are distributed as in Figure 5.29. Numbers on the image mark the general area where the robot collided with walls.

Chapter 6

Discussion

While the path following tasks in Chapter 5 did not achieve something that was not possible before, they did it in a significant new way. The combination of anticipation and abstraction was able to build useful representations in two different domains with few changes to the system. The achievements and contributions of this thesis will be discussed in this chapter.

6.1 Achievements

Using only an abstraction and anticipation network, a simulated robot successfully followed an approximately 20 meter long path. This is not yet a general localization system, but many of the important components are in place.

Figures 5.26 and 5.28 show that the prediction error gives a measure of how close a robot is to an original path. The simulated robot has much lower prediction error as it drives down a path similar to the path it was trained on. Given this, it should be possible to use abstraction and anticipation networks to address the problem posed in Section 4.6.

Even when exactly retracing a path, the prediction error sometimes spikes. In the case of the simulated robot, these spikes did not adversely affect its ability to follow the path. By using the predicted motor commands, the robot was able to follow the path very closely when starting from the same starting position. The prediction error was sometimes high enough that it would start a turn a few timesteps before or after the turn started in the training run, but the difference was minimal.

When starting from positions farther from the starting position, as in Figure 5.30, the robot is able to successfully navigate the path on many of the runs. At some points the robot became confused, and turned in ways which it never had during training. This is to be expected. The robot was only trained on a single run, and it therefore had no ability to learn how to react to going off course. One bad decision could effectively end a run.

We proposed using reinforcement learning to learn more robust control policies. We were able to show that prediction error can be used to generate a reward signal (see Figures 5.17 and 5.31), but the systems failed to take advantage of the information already stored in the prediction networks. As a result, it took very long to learn, and we failed to create robust control policies. Ideally, we would like to be able to learn a general procedure to follow when prediction error gets too high. People learn how to get back on track after becoming lost, and we would like our robots to do the same. This is a difficult problem, and attempting to solve it is beyond the scope of this thesis.

Another potential use of prediction error becomes apparent here. When prediction error is low, we expect predicted commands to be approximately correct. When prediction error is high, we should not have the same expectations. In such

situations it would be possible to take an action to reduce confusion. If nothing else, the robot could stop moving and notify a person of the problem. The spikes in prediction error may make it difficult to know when notification is necessary, but the idea has merit.

From what we have learned, it is possible to propose more interesting usage scenarios that just path following. For example, if we had sensor logs of the robot driving all possible paths from room to room in the simulated environment, an anticipation network could be trained for each of these paths. Then, as a person drove the robot through the environment, the prediction error would tell us how close we were to each path. When the prediction error for a path was sufficiently low, the robot could alert the driver that it thinks he is driving from *Room 1* to *Room 2*. However, there may be overlapping paths, and the prediction error from multiple networks might be low. In this case the robot could present the driver with a list of possibilities. In this way, anticipation networks could be used for activity recognition.

While it is interesting to be able to find out what a driver is doing, ultimately it does not help us much. However, after the driver confirms that he is in fact on a path the robot has proposed, we can actually turn over control from the driver to the robot. Provided that robust control policies could be learned, such a system could be extremely useful on a teleoperated robot.

6.2 Contributions

The results presented in the previous section are interesting to two different groups of researchers. The first group, developmental robotics researchers, is composed of people concerned with using developmental processes on robots. The second group, researchers attempting to learn navigation tasks, is composed of people who want their robots to learn how to navigate in real world environments. This thesis provides contributions to both groups that are best discussed separately.

6.2.1 Developmental Robotics

Work in developmental robotics is usually presented in a task-independent way. This thesis, however, has a specific goal. Unlike developmental robotics, the end goal here is not a general-purpose developmental learning algorithm, but a general localization algorithm. By first choosing what we want to achieve and then using developmental processes only as a tool, we avoided becoming too attached to a particular architecture. Instead, we molded and enhanced existing processes in order to make implicit localization possible.

By coming at the problem this way, we observed a flaw in the basic anticipation and abstraction model by Blank et al. (2005): the inability to directly use information contained in the more powerful, higher-level developmental layers. Indeed, previous work never extended to multiple layers. In order to work around this problem, the anticipation layer was given increased representational power by using multi-step instead of single-step prediction. Multi-step prediction enabled the prediction layer to understand a much wider variety of sequences. While this change

was made only to allow prediction to be useful for the long and complicated task of localization, it could be applied to a vast variety of other problems. The use of multi-step prediction is a contribution to the developmental robotics community.

6.2.2 Learned Robot Navigation

Many authors have applied machine learning to navigation tasks. From obstacle avoidance to goal-finding, there is a large body of work. Particularly relevant here is the work by Provost et al. (2006) and Smart and Kaelbling (2002). Both works considered goal finding, and the path following task in this thesis is very similar. In goal finding we want the robot to reach a specific goal, while in path following we want a robot to follow a specific path. In the environments used by Provost et al. and Smart and Kaelbling the environments were simple enough that there was only a single path to the goal, and in both cases the robot's task was to take this path.

Provost et al. (2006) developed a path-following robot that learned to associate sensor inputs directly with high-level actions. Abstraction was provided by a self-organizing map-like structure that converted sensor values to discrete states. Reinforcement learning was then used to determine the next action to take. New actions were only chosen when the discrete state abstraction changed. By choosing actions less often, the number of states that were visited in each training episode was greatly decreased, and training proceeded much faster.

Smart and Kaelbling (2002) used Q-Learning with locally weighted regression as a function approximation tool to perform a similar task. Bootstrapping was used much the same as in this work, where known correct runs were provided to the robot before it attempted to complete the task. The important contribution of their work

was a drastic increase in learning speed, as it could perform the task better than the examples after only a few episodes.

Both of the works described above performed similar tasks to this work, but in different ways. Both the approaches map sensor values directly to actions, so they could not be used in cases where similar states should be handled differently. However, they also learned more robust control policies than were shown in Chapter 5. As demonstrated in the previous chapter, predictive control was capable of duplicating behavior, but was incapable of dealing robustly with change. Offset starting positions in the simulated building quickly caused unrecoverable errors.

Although the reinforcement learning tests presented here did not achieve robust control, the ideas at work are useful. Particularly, the ability to generate a reward function based on prediction error could be very beneficial. Both Provost et al. (2006) and Smart and Kaelbling (2002) used reward functions based on global information. The need for global information makes their robots able to learn autonomously only when an accurate localization method is available, and the value of learning navigation based on a working localization algorithm is questionable. Therefore, the major contribution of this work to the community working on using machine learning for navigation tasks is a new way to create a reward function based on training examples. The process used here was not entirely automatic. We did not directly use the prediction error as a reward signal, but instead set a threshold and only rewarded the agent when prediction error was sufficiently low.

This is related to both imitation learning (Atkeson and Schaal, 1997) and inverse reinforcement learning (IRL) (Abbeel and Ng, 2004). Imitation learning systems attempt to imitate the behaviors demonstrated in training examples. It is possible to

perform this task without posing the problem in a reinforcement learning framework. On the other hand, IRL models the problem in a RL framework. IRL attempts to build a reward function that can be used to achieve "good" behavior, and training examples are used to define "good" behavior. The algorithm presented by Abbeel and Ng is general, but it expresses reward as a linear function of known environmental features. For a path following task it is unclear what features would be provided to the inverse reinforcement learning algorithm. Although it did not receive study here, it would be interesting to see if the internal representations learned by an anticipation network could be used to generate features for IRL.

Chapter 7

Conclusion

This thesis demonstrated the construction of a general system for performing robot localization and, to some extent, control. A simulated robot successfully learned to traverse a path that was more complex than published works using related methods.

7.1 Summary

Implicit localization has been successfully developed in simulated environments. Starting with no initial knowledge, and just a single run of training data, the robot is able to later recognize portions of the path. This is somewhat similar to the global localization problem where a lost robot has to reacquire its position. It was also clearly shown that as long as the simulated robots stayed close to their original paths, the anticipation network could be used both for prediction and control. This shows that the learned localization strategy had the ability to perform position tracking.

This work even meets some of the goals of simultaneous localization and mapping. That is, if a robot has a method to traverse a path, then it can learn a representation

for the path with no human interaction. However, it does not learn online or know how to explore its environment, so the problem is far from fully addressed. Still, the fact that the same system was able to understand paths through both a grid world and a simulated environment shows how generally applicable the system is. The same can not be said for heavily engineered human solutions.

Lastly, it was shown that the learned prediction network could be used as a source of information for another learning algorithm. Specifically, the prediction error was used as a reward signal for reinforcement learning. However, other combinations would also be possible. The automatic generation of reward functions may be useful for a wide variety of robotics tasks.

7.2 Future Work

There are several key areas that could use further study: the real-world applicability of the system, the possibility of using bootstrapping to improve performance, and the usefulness of prediction error for other purposes.

First, because tests were only run in simulation, it is difficult to say how well the presented system would transfer to the real world. Although both simulation environments included some degree of error, it is unknown how much real-world error the abstraction and anticipation networks would be able to handle.

Second, we would like to be able to use only a single training example, and then let the system create additional training examples that could increase its robustness. We have shown that it is possible to repeat a task based on a single training example. If we logged the sensorimotor information from successful task completions and

retrained the anticipation network to recognize the new data, then it might be possible to gradually expand the range of conditions that the system is able to understand. In such a way, it may be possible to start from a non-robust control system and use bootstrapping to increase the performance over time.

Third, the implementation of reinforcement learning was not ideal. It took much too long for the agents to reach acceptable performance. Ideally, we would like to have only tens of training runs instead of hundreds or thousands. The fact that predictive control worked so well showed that there was significant knowledge stored in the network. However, the Q-Learning based implementation failed to efficiently transfer the knowledge. Alternate reinforcement learning algorithms should be considered.

Reference List

Abbeel, P. and A. Y. Ng, 2004: Apprenticeship learning via inverse reinforcement learning. *Proceedings of the 21st International Conference on Machine Learning*.

Atkeson, C. G. and S. Schaal, 1997: Robot learning from demonstration. *Proceedings of the 14th International Conference on Machine Learning*, Morgan Kaufmann, 12–20.

Barto, A. G. and Ö. Şimşek, 2005: Intrinsic motivation for reinforcement learning systems. *Proceedings of the 13th Yale Workshop on Adaptive and Learning Systems*, 113–118.

Bertsekas, D. P. and J. N. Tsitsiklis, 1996: *Neuro-Dynamic Programming*. Athena Scientific.

Blank, D., D. Kumar, and L. Meeden, 2002: A developmental approach to intelligence. *Proceedings of the 13th Annual Midwest Artificial Intelligence and Cognitive Science Society Conference*.

Blank, D., D. Kumar, L. Meeden, and J. B. Marshall, 2005: Bringing up robot: Fundamental mechanisms for creating a self-motivated, self-organizing architecture. *Cybernetics and Systems*, **36**, 125–150.

Bui, H., S. Venkatesh, and G. West, 2002: Policy recognition in the abstract hidden markov model. *Journal of Artificial Intelligence Research*, **17**, 451–499.

Davison, A. J., 2003: Real-time simultaneous localisation and mapping with a single camera. *Proceedings of the 9th International Conference on Computer Vision*, 1403–1410.

Dellaert, F., D. Fox, W. Burgard, and S. Thrun, 1999: Monte Carlo localization for mobile robots. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1322–1328.

Duckett, T., S. Marsland, and J. Shapiro, 2002: Fast, on-line learning of globally consistent maps. *Autonomous Robots*, **12**, 287–300.

Eliazar, A. I. and R. Parr, 2005: Hierarchical linear/constant time SLAM using particle filters for dense maps. *Neural Information Processing Systems*.

Elman, J. L., 1990: Finding structure in time. *Cognitive Science*, **14**, 179–211.

Fahlman, S. E., 1988: Faster-learning variations on back-propagation: An empirical study. *Proceedings of the 1988 Connectionist Models Summer School*.

Gerkey, B., R. T. Vaughan, and A. Howard, 2003: The Player/Stage Project: Tools for multi-robot and distributed sensor systems. *Proceedings of the 11th International Conference on Advanced Robotics*, 317–323.

Hochreiter, S. and J. Schmidhuber, 1997: Long short-term memory. *Neural Computation*, **9**, 1735–1780.

Kohonen, T., 1984: *Self-organization and associative memory*. Springer-Verlag, Berlin, Germany.

Kramer, M. A., 1991: Nonlinear principal component analysis using autoassociative neural networks. *American Institute of Chemical Engineers Journal*, **37**, 233–243.

LeCun, Y., 1985: Une procédure d'apprentissage pour réseau a seuil asymmetrique (a learning scheme for asymmetric threshold networks). *Proceedings of Cognitiva 85*, 599–604.

Lungarella, M., G. Metta, R. Pfeifer, and G. Sandini, 2003: Developmental robotics: a survey. *Connection Science*, **15**, 151–190.

Madokoro, H., K. Sato, and M. Ishii, 2003: Acquisition of world images and self-localization estimation using viewing image sequences. *Systems and Computers in Japan*, **34**, 68–78.

McGovern, E. A., 2002: *Autonomous discovery of temporal abstractions from interaction with an environment*. Ph.D. thesis, Department of Computer Science, University of Massachusetts, Amherst, MA, USA.

Minsky, M. and S. Papert, 1969: *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA.

Mitchell, T. M., 1997: *Machine Learning*. McGraw-Hill, New York, NY, USA.

Montemerlo, M., S. Thrun, D. Koller, and B. Wegbreit, 2003: FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. *Proceedings of the 16th International Joint Conference on Artificial Intelligence*.

Nolfi, S. and J. Tani, 1999: Extracting regularities in space and time through a cascade of prediction networks: The case of a mobile robot navigating in a structured environment. *Connection Science*, **11**, 125–148.

Osentoski, S., V. Manfredi, and S. Mahadevan, 2004: Learning hierarchical models of activity. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Provost, J., P. Beeson, and B. J. Kuipers, 2001: Toward learning the causal layer of the spatial semantic hierarchy using SOMs. *AAAI Spring Symposium Series, Learning Grounded Representations*.

Provost, J., B. J. Kuipers, and R. Miikkulainen, 2004: Self-organizing perceptual and temporal abstraction for robot reinforcement learning. *AAAI Workshop on Learning and Planning in Markov Processes*.

Provost, J., B. J. Kuipers, and R. Miikkulainen, 2006: Developing navigation behavior through self-organizing distinctive state abstraction. *Connection Science (accepted for publication)*, **18**.

Riedmiller, M. and H. Braun, 1993: A direct adaptive method for faster backpropagation learning: The RPROP algorithm. *Proceedings of the IEEE International Conference on Neural Networks*, 586–591.

Rosenblatt, F., 1958: The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, **65**, 386–408.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams, 1986: Learning internal representations by error propagation. *Parallel distributed processing: explorations in the microstructure of cognition*, MIT Press, Cambridge, MA, USA, volume 1, 318–362.

Rumelhart, D. E., B. Widrow, and M. A. Lehr, 1994: The basic ideas in neural networks. *Communications of the ACM*, **37**, 87–92.

Samuel, A. L., 1959: Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 210–229, reprinted in E. A. Feigenbaum and J. Feldman, editors, Computers and Thought, McGraw-Hill, New York, 1963.

Şimşek, Ö. and A. G. Barto, 2006: An intrinsic reward mechanism for efficient exploration. *Proceedings of the 23rd International Conference on Machine Learning*.

Smart, W. D. and L. P. Kaelbling, 2002: Effective reinforcement learning for mobile robots. *Proceedings of the IEEE International Conference on Robotics and Automation*, 3404–3410.

Sutton, R. S., 1988: Learning to predict by the methods of temporal differences. *Machine Learning*, **3**, 9–44.

Sutton, R. S. and A. G. Barto, 1998: *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA.

Sutton, R. S., D. Precup, and S. P. Singh, 1999: Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, **112**, 181–211.

Tesauro, G., 1995: Temporal difference learning and TD-Gammon. *Communications of the ACM*, **38**, 58–68.

Thrun, S., 2002: Robotic mapping: A survey. *Exploring Artificial Intelligence in the New Millenium*, G. Lakemeyer and B. Nebel, eds., Morgan Kaufmann, 1–36.

Vaughan, R., K. Stoy, G. Sukhatme, and M. Mataric, 2002: LOST: Localization-space trails for robot teams. *IEEE Transactions on Robotics and Automation*, **18**, 796–812.

Watkins, C. J. C. H., 1989: *Learning from delayed rewards*. Ph.D. thesis, King's College of Cambridge, UK.

Werbos, P., 1974: *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA, USA.

Williams, R. J. and D. Zipser, 1989: A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, **1**, 270–280.