Memory Approaches To Reinforcement Learning In Non-Markovian Domains

Long-Ji Lin Tom M. Mitchell

May 1992
CMU-CS-92-138

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Reinforcement learning is a type of unsupervised learning for sequential decision making. *Q-learning* is probably the best-understood reinforcement learning algorithm. In Q-learning, the agent learns a mapping from states and actions to their utilities. An important assumption of Q-learning is the Markovian environment assumption, meaning that any information needed to determine the optimal actions is reflected in the agent's state representation. Consider an agent whose state representation is based solely on its immediate perceptual sensations. When its sensors are not able to make essential distinctions among world states, the Markov assumption is violated, causing a problem called *perceptual aliasing*. For example, when facing a closed box, an agent based on its current visual sensation cannot act optimally if the optimal action depends on the contents of the box. There are two basic approaches to addressing this problem— using more sensors or using history to figure out the current world state. This paper studies three connectionist approaches which learn to use history to handle perceptual aliasing: the *window-Q*, *recurrent-Q*, and *recurrent-model architectures*. Empirical study of these architectures is presented. Their relative strengths and weaknesses are also discussed.

This research was supported in part by Fujitsu Laboratories Ltd and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Fujitsu Laboratories Ltd or the U.S. Government.

Keywords: reinforcement learning, Markov/non-Markov decision task, action model, time-delay neural network, recurrent neural network

Contents

1 Introduction **1**

2 Perceptual Aliasing **2**

3 Three Memory Architectures **2**

4 The OAON Architectures **5**

5 Experimental Designs and Results **8**

5.1 Task 1: 2-Cup Collection 8

5.2 Task 2: Task 1 With Random Features 12

5.3 Task 3: Task 1 With Control Errors 12

5.4 Task 4: Pole Balancing 14

6 Discussion **16**

6.1 Architecture Characteristics 17

7 Related Work **19**

8 Conclusion **21**

A The Experience Replay Algorithm **22**

B The Parameter Settings For The Experiments **24**

C The Pole Balancing Problem **25**

1 Introduction

In the reinforcement learning paradigm, a learning agent continually receives inputs from its sensors, determines its action based on the inputs and its control policy, executes that action, and receives a scalar *reinforcement* or *payoff*. The goal of the agent is to construct an optimal policy so as to maximize its performance, which is often measured by the *discounted cumulative reinforcement* through the future (or for short, *utility*). The best-understood reinforcement learning algorithm is probably *Q-learning* [Watkins, 1989] [Lin, 1992]. The idea of Q-learning is to construct a Q-function:

$$Q(state, action) \rightarrow utility \tag{1}$$

The Q-function is used to predict the discounted cumulative reinforcement (also called utility or Q-value) for each state-action pair given that the agent is in that state and executes that action. Given a Q-function, the control policy is simply to choose the action a for which $Q(x, a)$ is maximal. The Q-function is incrementally constructed based on *temporal difference* (TD) *methods* [Sutton, 1988], which can be related to dynamic programming [Barto *et al.*, 1991]. Given a state transition (x, a, y, r) meaning that an action a in response to a state x results in a new state y and immediate payoff r, the Q-function can be updated in the following way:

$$Q(x,a) \leftarrow Q(x,a) + \eta \cdot (r + \gamma \cdot \max_k Q(y,k) - Q(x,a)) \tag{2}$$

where γ is a discount factor and η is the learning rate. Note that this update rule is based on TD(λ) with $\lambda = 0$. The update rule used in this work was more sophisticated than (2) and used $\lambda > 0$. For a detailed description of the algorithm, see Appendix A and [Lin, 1991] [Lin, 1992].

Watkins has shown that Q-learning will converge and find the optimal policy under two primary conditions (along with a few weak ones): (1) a look-up table representation is used for the Q-function, and (2) the environment is Markovian. The latter means that at any given time, the next state of the environment is determined only by the current state and the action taken. In such environments, all information needed to determine the current optimal action is reflected in the current state representation. Although the first condition disallows generalization, when the state space is large and the look-up table method is unacceptable, some generalization method is often used in modeling the Q-function. For example, Lin [Lin, 1991] [Lin, 1992] has successfully combined the connectionist error back-propagation algorithm and Q-learning to solve several nontrivial learning problems.

Consider a reinforcement learning agent whose state representation is based on only its immediate sensation. When its sensors are not able to make essential distinctions among world states, the Markov assumption mentioned above is violated, causing a problem called *perceptual aliasing* [Whitehead & Ballard, 1991]. For example, consider a packing task which involves 4 steps: open a box, put a gift into it, close it, and seal it. An agent driven only by its current visual percepts cannot accomplish this task, because when facing a closed box, the agent does not know if a gift is already in the box and therefore cannot decide whether to seal or open the box. There are two solutions to this problem. The first solution is to actively choose another perceptual action, such as measuring the weight of the box, to resolve

the ambiguity. The second solution is to use history information, such as whether a gift was ever put in the box, to help determine the current world state.

There is some previous work studying the first kind of solutions [Whitehead & Ballard, 1991] [Tan, 1991]. In this paper we focus on the second kind of solutions. The rest of the paper is organized as follows. First we distinguish two types of perceptual aliasing, followed by our three connectionist architectures for the problem. An OAON network architecture is also proposed for model learning and Q-learning. Empirical studies of these architectures are presented, and their relative strengths and weaknesses are also discussed. Section 7 discusses related work.

2 Perceptual Aliasing

We distinguish two types of perceptual aliasing: *voluntary* and *involuntary* [Chrisman, personal communication]. Consider an agent with a set of sensing operations which allow the agent to make complete distinctions among world states. Because some sensing operations are very costly (due to either time constraints or physical resource constraints), the agent might choose to apply only some subset of its available sensing operations. If its choice of limited sensing operations tends to perceptual aliasing which would not occur using all available operations, then we call this voluntary perceptual aliasing.

[Whitehead & Ballard, 1991] proposed a *lion* algorithm, which learns to focus perceptual attention in order to collect necessary sensory information for optimal control. [Tan, 1991] proposed a *cost-sensitive* algorithm, which learns to choose minimal sensing operations to disambiguate world states. These two algorithms might be called *sensor approaches* to (voluntary) perceptual aliasing. In contrast, the approaches presented here may be characterized as memory-based.

Involuntary perceptual aliasing is due to limitations of sensors available to an agent. For example, the color of the object in a closed box cannot be identified by vision. If the optimal action depends on that color, the agent must rely on its memory rather than on its sensors. Using memory or history information to control a non-Markovian environment will be referred to as memory approach to perceptual aliasing. Note that some voluntary perceptual aliasing can also be handled through memory approaches. On the other hand, there are situations where memory approaches fail. Consider the example in the beginning of this paragraph. If the agent never got a chance to see the color of the object before it was packed in the box, memory approaches will certainly fail in this situation. To deal with complex real-world problems efficiently, both sensor and memory approaches will be needed. Integration of both types of approaches remains to be investigated.

3 Three Memory Architectures

Figure 1 depicts three memory architectures for reinforcement learning in non-Markovian domains. One is a type of indirect control, and the other two direct control. These architectures

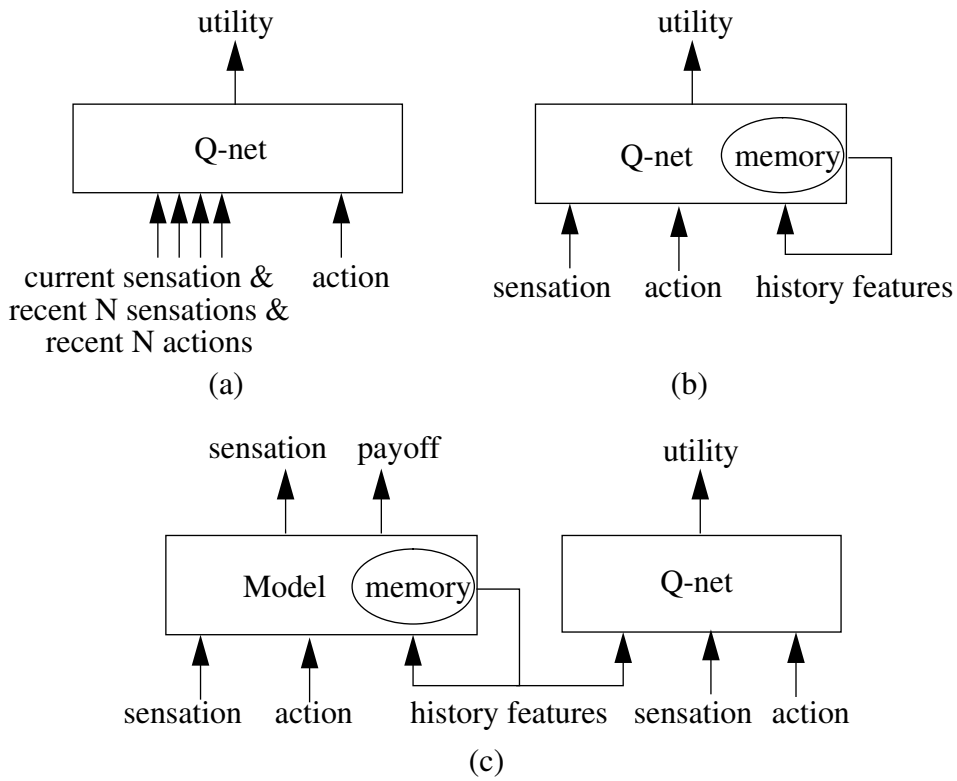

Figure 1: Three memory architectures for reinforcement learning in non-Markovian domains: (a) window-Q architecture, (b) recurrent-Q architecture, and (c) recurrent-model architecture.

all use temporal difference methods to incrementally learn a Q-function, which is represented with neural networks.

Instead of using just the current sensation as state representation, the *window-Q architecture* uses the current sensation, the N most recent sensations, and the N most recent actions taken all together as state representation. In other words, the window-Q architecture allows a direct access to the information in the past through a sliding window. N is called the *window size*. The window-Q architecture is simple and straightforward, but the problem is that we may not know the right window size in advance. If the window size is chosen to be not large enough, there will be not enough history information to construct an optimal Q-function. Moreover, if the window size needs to be large, the large number of units in the input layer will require a lot of training patterns for successful learning and generalization. In spite of these problems, it is still worth study, since this kind of *time-delay neural networks* has been found quite successful in speech recognition [Waibel, 1989] and several other domains.

The window-Q architecture is sort of a brute force approach to using history information. An alternative is to distill a (small) set of *history features* out of the large amount of information in the past. This set of features together with the current sensation become the agent's state

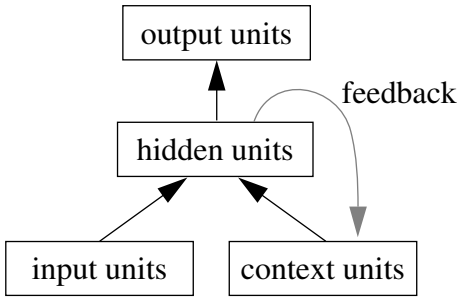

Figure 2: An Elman Network.

representation. The optimal control actions then can be determined based on just this new state representation, if the set of history features reflects the information needed for optimal control. The *recurrent-Q* and *recurrent-model* architectures illustrated in Figure 1 are based on this same idea, but the ways they construct history features are different. Unlike the window-Q architecture, both architectures in principle can discover and utilize history features that depend on sensations arbitrarily deep in the past, although in practice it is difficult to achieve this.

Recurrent neural networks, such as Elman networks [Elman, 1990], provide a way to construct history features. As illustrated in Figure 2, the input layer of an Elman network is divided into two parts: the true input units and the *context units*. The context units hold feedback signals coming from the network state at a previous time step. The context units, which function as the memory in Figure 1, remember an aggregate of previous network states, and so the output of the network depends on the past as well as on the current input.

The recurrent-Q architecture uses a recurrent network to model the Q-function. To predict utility values correctly, the recurrent network (called recurrent Q-net) will be forced to learn history features which enable the network to properly assign different utility values to states with the same sensation.

The recurrent-model architecture consists of two concurrent learning components: learning an *action model* and Q-learning. An action model is a function which maps a sensation and an action to the next sensation and the immediate payoff. To predict the behavior of the environment, the recurrent action model will be forced to learn a set of history features. Using the current sensation and the set of history features as state representation, we can turn a non-Markovian task into Markovian one and solve it using the conventional Q-learning, if a perfect action model is available. This is simply because at any given time, the next state of the environment now can be completely determined by this new state representation and the action taken.

Both the recurrent-Q and recurrent-model architectures learn history features using a gradient descent method (e.g., error back-propagation), but they differ in an important way. For model learning, the goal is to minimize the errors between actual and predicted sensations and payoffs.

In essence, the environment provides all the needed training information, which is consistent over time as long as the environment does not change. For recurrent Q-learning, the goal is to minimize the errors between *temporally successive predictions* of utility values [Sutton, 1988] [Lin, 1992]. The error signals here are computed based partly on information from the environment and partly on the current approximation to the true Q-function. The latter changes over time and carries no information at all in the beginning of learning. In other words, these error signals are in general weak, noisy and even inconsistent over time. Because of this, whether the recurrent-Q architecture will ever work in practice may be questioned.

In general the action model must be trained to predict not only the new sensation but also the immediate payoff. Consider another packing task which involves 3 steps: put a gift into an open box, seal the box so that it cannot be opened again, place the box in the proper bucket depending on the color of gift in the box. A reward is given only when the box is placed in the right bucket. Note that the agent is never required to know the gift color in order to predict future sensations, since the box cannot be opened once sealed. Therefore a model which only predicts sensations will not have the right features which are needed to accomplish this task. However, for each of the tasks described in Section 5, the action model does not need to predict the immediate payoff in order to discover history features needed for optimal control.

It is worthwhile to note that combinations of the three architectures are possible. For example, we can combine the first two architectures: the inputs to the recurrent Q-net could include not just the current sensation but also recent sensations. We can also combine the last two architectures: the memory is shared between a recurrent model and a recurrent Q-net, and history features are developed using the error signals coming from both the model and the Q-net. This paper is only concerned with the three basic architectures. Further investigation is needed to see if these kinds of combination will result in better performance than the basic versions.

4 The OAON Architectures

Both the Q-function and action model take two kinds of inputs: sensation and action. There are two alternative network structures to realize them. One is a monolithic network whose input units encode both the sensation and the action. For domains with discrete actions, this structure is often undesirable, because the monolithic network is to be trained to model a highly nonlinear function— given the same sensation (and same history features), different actions may demand very different outputs of the network.

Another alternative is what we call the OAON (*One Action One Network*) architectures. We use multiple networks, one for each action, to represent the Q-function and the model. Figure 3 illustrates two types of recurrent OAON architectures: linear and nonlinear. The linear OAON consists of single-layer perceptrons, and the nonlinear one consists of multilayer networks of units with a nonlinear squashing function. At any given time, only the network corresponding to the selected action is used to compute the next values of the output and context units, while the others can be ignored. It is important to note that we also have to ensure that the multiple networks will use the same (distributed) representation of history features. This is achieved

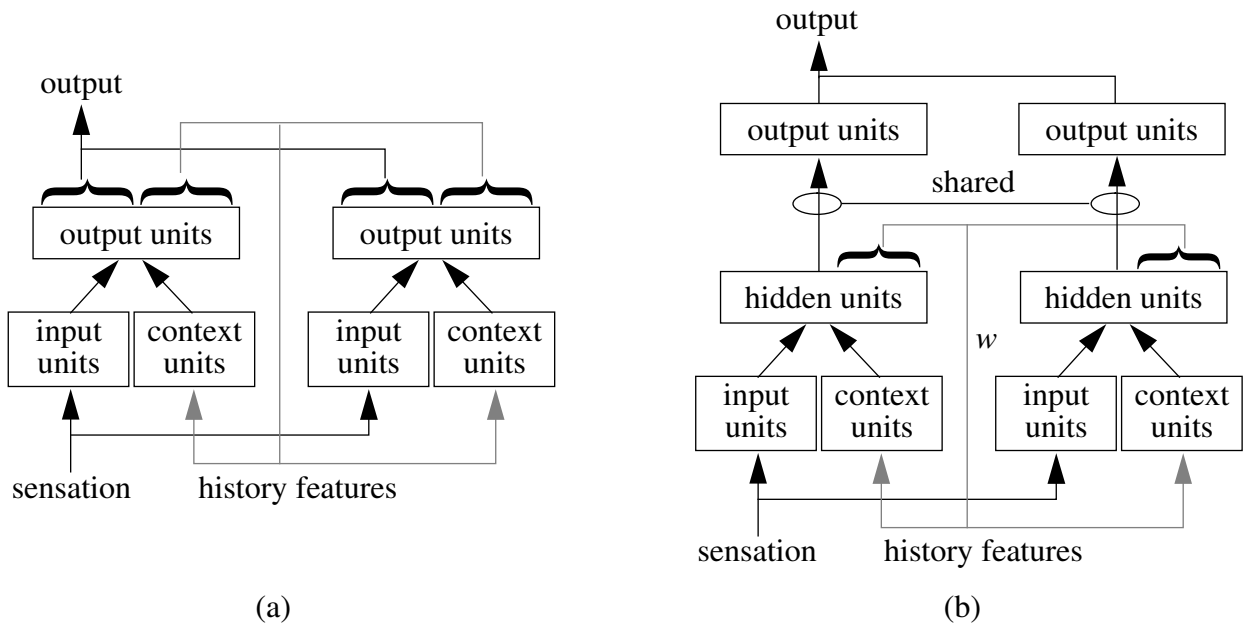

Figure 3: Two OAON architectures for recurrent models and recurrent Q-nets: (a) linear and (b) nonlinear. The linear one consists of multiple (here, 2) perceptrons, and the nonlinear one consists of (modified) Elman networks. In both cases, at any given time, only the network corresponding to the selected action is used to compute the next values of the output and context units.

by having the networks share activations of context units, and can be further reinforced (in the case of the nonlinear one) by having the networks share all connections emitting from the hidden layer. As found empirically, the sharing of connections does not seem necessary but tends to help.

As shown in Figure 3, the input layer of each network (either linear or nonlinear) is divided into two parts: the true input units and the context units, as we have seen in the Elman network (Figure 2). Note that each of the recurrent networks in Figure 3.b is a modified Elman network. There are three differences between the "standard" Elman network and our modified version. First the Elman network does not use *back-propagation through time*, but ours does. (Also see below.) Second, the whole hidden layer of the Elman network is fed back to the input layer, but here only a portion of hidden units is fed back. Consider an environment with little perceptual aliasing; in other words, feed-forward networks are sufficient to model most aspects of the environment, and just one history feature needs to be discovered in order to model the whole environment. In such a case, it makes sense to have many hidden units but just one context unit. From our limited experience, the Elman network tended to need more context units than this modified one. As a consequence, the former normally required more connections than the latter. Furthermore, since the context units are also part of the inputs to the Q-net, more context units require more training patterns and training time on

the part of Q-learning. The third difference is in the constant feedback weight, the w in Figure 3.b. The Elman network uses $w = 1$, while we found that w slightly greater than 1 (say, 1.5) tends to make the network converge sooner. The reason it might help is the following: In the early phase of training, the activations of context units are normally close to zero (we used a symmetric squashing function). Therefore, the context units appear unimportant compared with the inputs. By magnifying their activations, the back-propagation algorithm will pay more attention to the context units.

The linear OAON architecture shown in Figure 3.a is in fact the SLUG architecture proposed by [Mozer & Bachrach, 1991], who have applied SLUG to learn to model finite state automata (FSA) and demonstrated its success for several domains. (They did not use SLUG to model a Q-function, however). They also found that the conventional recurrent networks, such as the Elman network, were "spectacularly unsuccessful" at modeling FSA's. This may be explained by the fact that their experiments took the monolithic approach (one network for all actions). In contrast, our experiments using the nonlinear OAON architecture were quite successful. It always learned perfectly the FSA's that SLUG was able to learn, although it seemed that more training data would be needed to acquire perfect models due to the fact that nonlinear networks often have more degrees of freedom (connections) than linear ones.[1]

To model FSA's, we probably do not need a nonlinear OAON; linear OAONs (SLUG) may suffice and outperform nonlinear OAONs. But for many applications, a nonlinear OAON is necessary and cannot be replaced by the linear one. For example, Q-functions are in general nonlinear. For a pole balancer to be discussed later, its action model is also nonlinear. For the sake of comparing results of our various experiments, we used nonlinear OAON for all the recurrent Q-nets and models in all the experiments to be presented below.

Both the hidden units and the output units (in the nonlinear case) used a symmetric sigmoid squashing function $y = 1/(1 + e^{-x}) - 0.5$. A slightly modified version of the back-propagation algorithm [Rumelhart *et al.*, 1986] was used to adjust network weights. *Back-propagation through time* (BPTT) [Rumelhart *et al.*, 1986] was used and found to be significant improvement over back-propagation without BPTT. To apply BPTT, the recurrent networks were completely unfolded in time, errors were then back-propagated through the whole chain of networks, and finally the weights were modified according to the cumulative gradients. To restrict the influence of output errors at time t on the gradients computed at times much earlier than t, we applied a decay to the errors when they were propagated from the context units at time t to the hidden layer at time $t - 1$. Better performance was found with this decay than without it. The decay we used here was 0.9. The context units were initialized to 0 at the start of each task instance.

[1] We must clarify what we mean by being able to learn a perfect model. Given any finite set of input-output patterns generated by an unknown finite state automaton (FSA), there are infinite number of FSA's which can fit perfectly the training data. Because of this, at any given moment, a model learning algorithm can never be sure whether it has learned exactly the same FSA as the one producing the training data, unless some characteristic (for example, the upper bound of size) of the target FSA is known in advance. Therefore, when we say our architectures learn some FSA perfectly, we mean that it can predict an environment perfectly for a long period of time, say a thousand of steps.

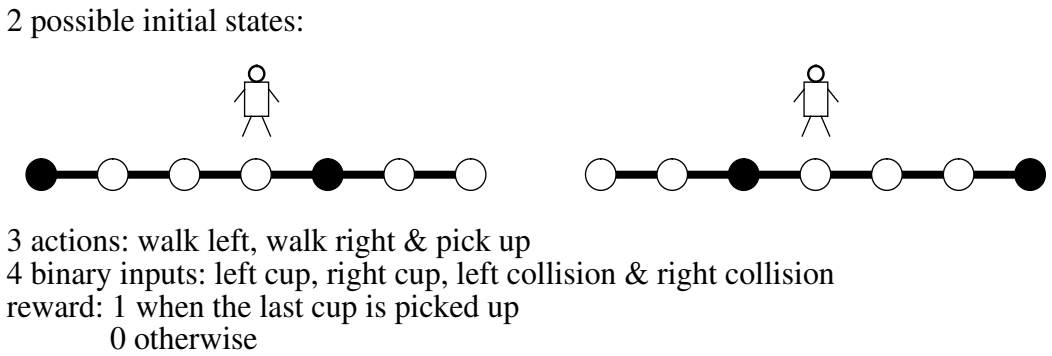

Figure 4: Task 1: A 2-cup collection task.

5 Experimental Designs and Results

In this section we describe our study of the three architectures for solving various learning problems with different characteristics. Through the study, we expect to gain more insights into these architectures, such as whether these architectures work and which one may work best for which types of problems.

Several parameters were used in the experiments, such as the discount factor γ, the recency factor λ used in TD(λ) methods, the learning rate, the number of context units, the number of hidden units, the window size, etc. Several of them, however, were fixed for all of the experiments. Appendix B gives a detailed description of the parameter settings we used, which were chosen to give roughly the best performance.

The *experience replay* algorithm described in Appendix A was used to significantly reduce the number of trials needed to learn an optimal control policy. By experience replay, the learning agent remembers its past action sequences and repeatedly applies its learning algorithm to each sequence in chronologically backward order. In all of the experiments here, only the experience from the most recent 70 trials was replayed to train the Q-function, while the action model was trained on all the experience in the past.

Each experiment consisted of two interleaved phases: a learning phase and a test phase. In the learning phase, the agent chose actions stochastically based on a Boltzmann distribution [Lin, 1992]. This randomness in action selection ensured sufficient exploration by the agent during learning. In the test phase, the agent always took the best actions according to its current policy. The learning curves shown below describe performance in the test phase.

5.1 Task 1: 2-Cup Collection

We started with a simple 2-cup collection task (Figure 4). This task requires the learning agent to pick up two cups located in a 1-D space. The agent has 3 actions: walking right one

cell, walking left one cell, and pick-up. When the agent executes the pick-up action, it will pick up a cup if and only if the cup is located at the agent's current cell. The agent's sensation includes 4 binary bits: 2 bits indicating if there is a cup in the immediate left or right cell, and 2 bits indicating if the previous action results in a collision from the left or the right. An action attempting to move the agent out of the space will cause a collision.

The cups are placed far enough apart that once the agent picks up the first cup, it cannot see the other one. To act optimally, the agent has to somehow remember the location of the second cup. This task is not trivial for several reasons: 1) the agent cannot sense a cup in front of it, 2) the agent gets no reward until both cups are picked up, and 3) the agent often operates with no cup in sight especially after picking up the first cup. Note that we restrict the problem such that there are only two possible initial states as shown in Figure 4. The reason for this restriction is to simplify the task and to avoid perceptual aliasing in the very beginning where no history information is available. The optimal policy requires 7 steps to pick up the 2 cups in each situation.

Figure 5 shows the learning curves for the three architectures. These curves show the mean performance over 5 runs with different seeds for the random number generator. The Y axis indicates the number of steps to pick up the four cups in both task instances shown in Figure 4 before time out (i.e., 30 steps). The X axis indicates the number of trials the agent has attempted so far. In each trial the agent started with one of the two possible initial states, and stopped either when both cups were picked up, or else after time out.

We experimented with two different values for the parameter λ. Two things are obvious from the learning curves. First, with $\lambda = 0.8$, all of these architectures successfully learned the optimal policy. Second, $\lambda = 0.8$ gave much better performance than $\lambda = 0$. Consider the situation in Figure 6, where A–E are five consecutive states and F is a bad state that displays the same sensation as State D. With $\lambda = 0$, the TD method estimates the utility of State C (for example) based on just the immediate payoff Rc and the utility of State D. Since State D displays the same sensation as the bad state F, the utility of State D will be much underestimated, until features are learned to distinguish States D and F. Before the features are learned, the utilities of State C and the preceding states will also much underestimated. This problem is mitigated when $\lambda > 0$ is used, because the TD($\lambda > 0$) method estimates the utility of State C not only based on Rc and State D, but also based on all the following rewards (Rd, Re, etc) and states (State E, etc).

The following are some additional observations, which are not shown in Figure 5:

The window size N. The performance in Figure 5.a was obtained with $N = 5$. As a matter of fact, the optimal policy could be learned (but only occasionally) with $N = 2$. Imagine that the agent has picked up the first cup and is walking towards the second cup. For a few steps, the most recent 3 sensations (including the current one) in fact do not provide the agent any information at all (namely, all sensation bits are off). How could the agent ever learn the optimal policy with $N = 2$? The way it worked is the following: After picking up the first cup, the agent determines the right direction to move. Later on, the agent simply follows the general direction the previous actions has been headed for. In other words, *the agent's action choices are themselves used as a medium for passing information from the past to the present.*

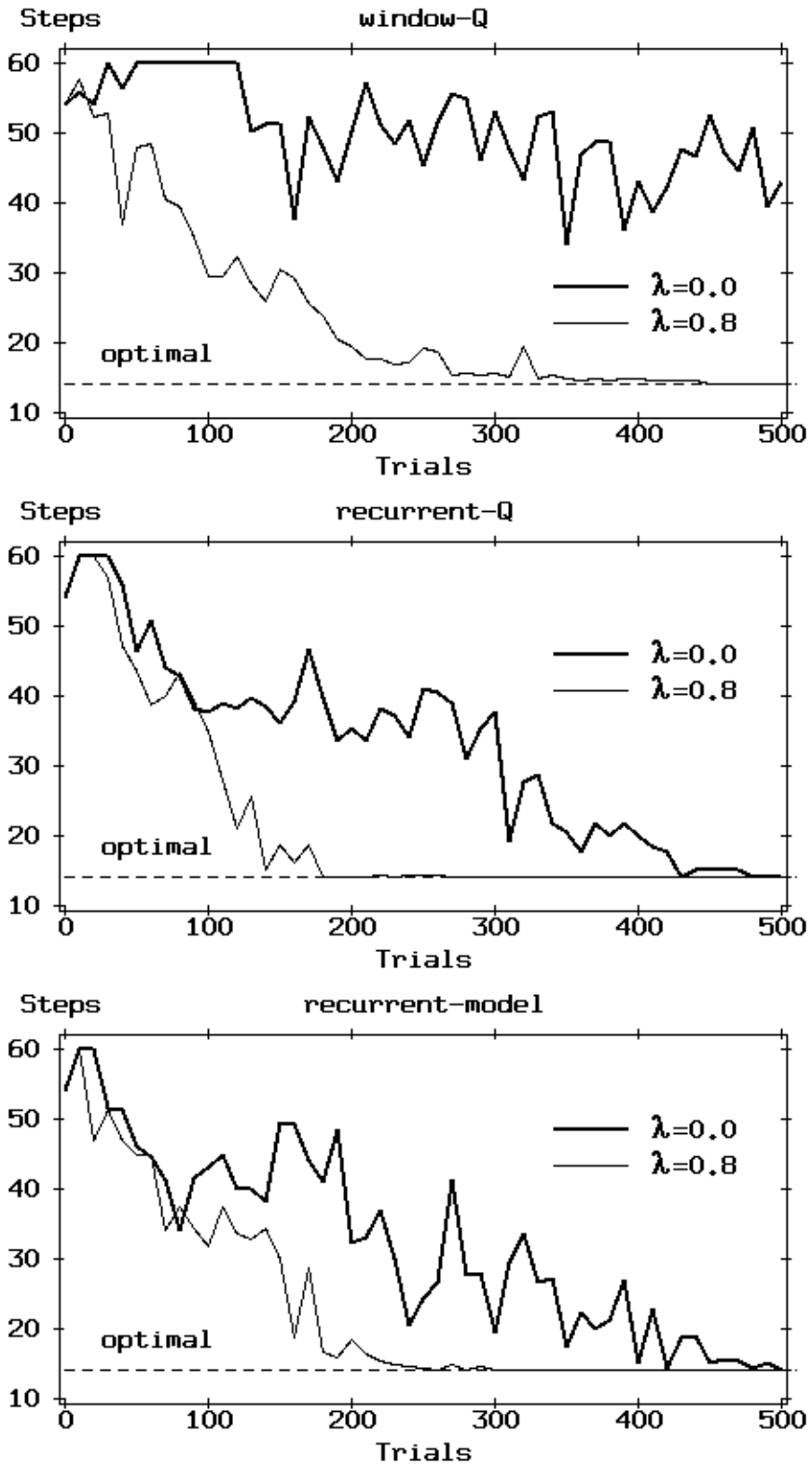

Figure 5: Performance for Task 1: (a) the window-Q architecture, (b) the recurrent-Q architecture, and (c) the recurrent-model architecture.

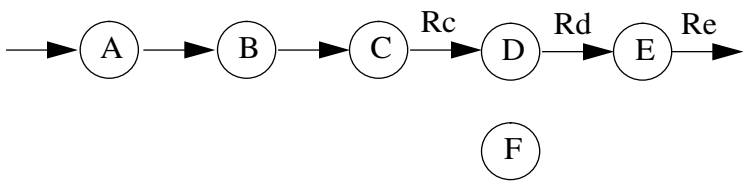

Figure 6: A difficult situation for TD(0). A–E are five consecutive states and F is a bad state. Rc, Rd, and Re are the payoffs received on the state transitions. Also, States D and F display the same sensation.

Robustness. The policy learned by the window-Q architecture is not very "robust", because we can fool it in the following manner: Suppose the agent is following its learned optimal policy. We suddenly interrupt the agent and force it to move in the opposite direction for a few steps. The agent then cannot resume the cup collection task optimally, because its policy will choose to continue on the new direction, which is opposite to the optimal direction (see the last paragraph). In contrast, the policy learned by the recurrent-model architecture was found more robust; it appeared to choose actions based on the actual world state rather than on the actions taken before. We also examined the policy learned by the recurrent-Q architecture. It appeared that the recurrent-Q architecture learned a policy somewhere between— sometimes it appeared to choose actions based on the actual world state, and sometimes based on its recent action choices. It would be instructive to see what history features were learned in the model versus in the recurrent Q-net. We plan to study this in the future.

Imperfect model. The agent never learned a perfect model within 500 trials. For instance, if the agent has not seen a cup for 10 steps or more, the model normally is not able to predict the appearance of the cup. But this imperfect model did not prevent Q-learning from learning an optimal policy. This reveals that *the recurrent-model architecture does not need to learn a perfect model in order to learn an optimal policy.* It needs to learn only the important aspects of the environment. But since it does not know in advance what are and what are not important to the task, it will end up attempting to learn everything including the details of the environment that are relevant to predicting the next sensation but unnecessary for optimal control.

Computation time. Both the recurrent-Q and recurrent-model architectures found the optimal policy after about the same number of trials, but the actual CPU time taken was very different. The former took 20 minutes to complete a run, while the latter took 80 minutes. This is because the recurrent-model architecture had to learn a model as well as a Q-net and the model network was much bigger than the recurrent Q-net.

This experiment revealed two lessons:

- All of the three architectures worked for this simple problem.
- For the recurrent-model architecture, just a partially correct model may provide sufficient history features for optimal control. This is good news, since a perfect model is often

difficult to obtain.

5.2 Task 2: Task 1 With Random Features

Task 2 is simply Task 1 with two random bits in the agent's sensation. The random bits simulate two difficult-to-predict and irrelevant features accessible to the learning agent. In the real world, there are often many features which are difficult to predict but fortunately not relevant to the task to be solved. For example, predicting whether it is going to rain outside might be difficult, but it does not matter if the task is to pick up cups inside. The ability to handle difficult-to-predict but irrelevant features is important for a learning system to be practical.

Figure 7 shows the learning curves of the three architectures for this task. Again, these curves are the mean performance over 5 runs. As we can see, the two random features gave little impact on the performance of the window-Q architecture or the recurrent-Q architecture, while the negative impact on the recurrent-model architecture was noticeable.

The recurrent-model did find the optimal policy many times after 300 trials. It just could not stablize on the optimal policy; it oscillated between the optimal policy and some sub-optimal policies. We also observed that the model tried in vain to reduce the prediction errors on the two random bits. There are two possible explanations for the poorer performance compared with that obtained when there are no random sensation bits. First, the model might fail to learn the history features needed to solve the task, because much of the effort was wasted on the random bits. Second, because the activations of the context units were shared between the model network and the Q-net, a change to the representation of history features on the model part could simply destabilize a well-trained Q-net, if the change was significant. The first explanation is ruled out, since the optimal policy indeed was found many times. To test the second explanation, we fixed the model at some point of learning and allowed only changes to the Q-net. In such a setup, the agent found the optimal policy and indeed stuck to it.

This experiment reveals two lessons:

- The recurrent-Q architecture is more economic than the recurrent-model architecture in the sense that the former will not try to learn a history feature if it does not appear to be relevant to predicting utilities.
- A potential problem with the recurrent-model architecture is that changes to the representation of history features on the model part may cause instability on the Q-net part.

5.3 Task 3: Task 1 With Control Errors

Noise and uncertainty prevail in the real world. To study the capability of these architectures to handle noise, we added 15% control errors to the agent's actuators, so that 15% of the time the executed action would not have any effect on the environment. (The 2 random bits were removed.) Figure 8 shows the mean performance of these architectures over 5 runs. Note that

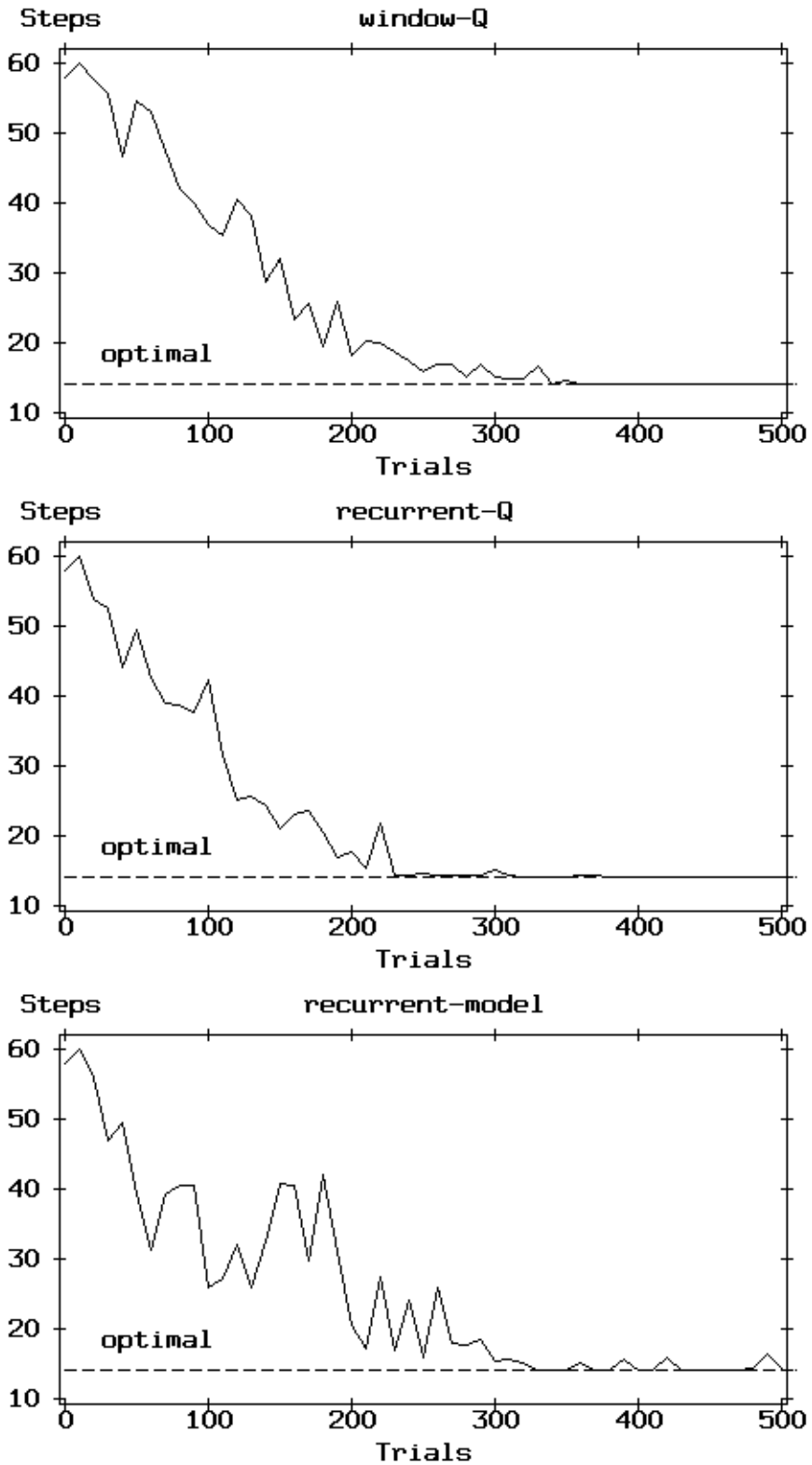

Figure 7: Performance for Task 2: (a) the window-Q architecture, (b) the recurrent-Q architecture, and (c) the recurrent-model architecture.

we turned off the control errors when we tested the performance of the agent, so the optimal number of steps in the figure is still 14. [2]

In 3 out of the 5 runs, the window-Q architecture successfully found the optimal policy, while in the other two runs, it only found suboptimal policies. In spite of the control errors, the recurrent-Q architecture always learned the optimal policy (with little instability).

The recurrent-model architecture always found the optimal policy after 500 trials, but again its policy oscillated between the optimal one and some sub-optimal ones due to the changing representation of history features, much as happened in Task 2. If we can find some way to stablize the model (for example, by gradually decreasing the learning rate to 0 at the end), we should be able to obtain a stable and optimal policy.

In short, we learned two lessons from this experiment:

- All of the three architectures can handle small control errors to some degree.
- Among the architectures, recurrent-Q seems to scale best in the presence of control errors.

5.4 Task 4: Pole Balancing

The traditional pole balancing problem is to balance a pole on a cart given the cart position, pole angle, cart velocity, and pole angular velocity. (See Appendix C for the detailed description of the problem.) This problem has been studied many times (e.g., [Anderson, 1987]) as a nontrivial control problem due to sparse reinforcement signals, which are -1 when the pole falls past 12 degrees and 0 elsewhere. Task 4 is the same problem except that only the cart position and pole angle are given to the learning agent. To balance the pole, the agent must learn features like velocity. In this experiment, a policy was considered satisfactory whenever the pole could be balanced for over 5000 steps in each of the 7 test trials where the pole starts with an angle of 0, ± 1, ± 2, or ± 3 degrees. (The maximum initial pole angle with which the pole can be balanced indefinitely is about 3.3 degrees.) In the training phase, pole angles and cart positions were generated randomly. The initial cart velocity and pole velocity are always set to 0. We used $N = 1$ in this experiment.

The input representation we used was straightforward: one real-valued input unit for each of the pole angle and cart position. Table 1 shows the number of trials taken by each architecture before a satisfactory policy was learned. These numbers are the average of the results from the best 5 out of 6 runs. (A satisfactory policy was not always found within 1000 trials.). In contrast, memoryless Q-learning took 119 trials to solve the traditional pole balancing problem, in which the agent is given the cart position, pole angle, cart velocity, and pole angular velocity.

One lesson was learned from this experiment:

[2] In this paper, we did not experiment with sensing errors, which will be future work. As pointed out in [Bachrach, 1992], to train a recurrent model properly in noisy environments, we may need to alter the learning procedure slightly. More precisely, current sensations cannot be 100% trusted for predicting future sensations. Instead, we should trust the model's predicted sensations more as the model gets better.

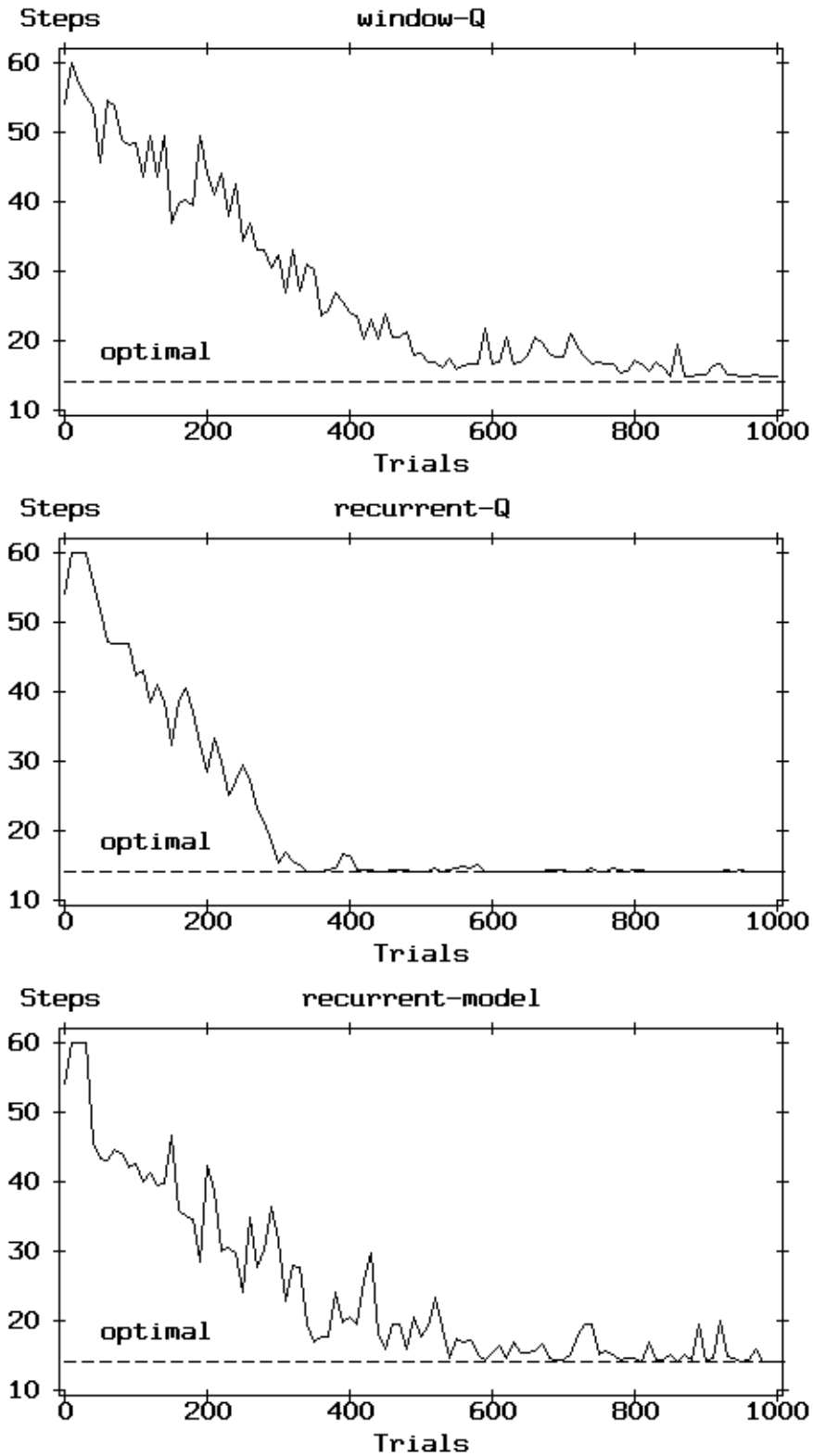

Figure 8: Performance for Task 3: (a) the window-Q architecture, (b) the recurrent-Q architecture, and (c) the recurrent-model architecture.

Table 1: Performance for the pole balancing task.

method	window-Q	recurrent-Q	recurrent-model
# of trials	206	552	247

- While the recurrent-Q architecture was the most suitable architecture for the cup collection tasks, it was outperformed by the other two architectures for the pole balancing task.

6 Discussion

The above experiments provide some insight into the performance of the three memory architectures. In this section, we consider problem characteristics that determine which architecture is most appropriate to which task environments. The three architectures exhibit different advantages whose relative importance varies with task parameters such as:

- **Memory depth.** One important problem parameter is the length of time over which the agent must remember previous inputs in order to represent an optimal control policy. For example, the memory depth for Task 1 is 2, as evidenced by the fact that the window-Q agent was able to obtain the optimal control based only on a window of size 2. The memory depth for the pole balancing task is 1. Note that learning an optimal policy may requires a larger memory depth than that needed to represent the policy.
- **Payoff delay.** In cases where the payoff is zero except for the goal state, we define the payoff delay of a problem to be the length of the optimal action sequence leading to the goal. This parameter is important because it influences the overall difficulty of Q-learning. As the payoff delay increases, learning an accurate Q-function becomes increasingly difficult due to the increasing difficulty of credit assignment.
- **Number of history features to be learned.** In general, the more perceptual aliasing an agent faces, the more history features the agent has to discover, and the more difficult the task becomes. In general, predicting sensations (i.e., a model) requires more history features than predicting utilities (i.e., a Q-net), which in turn requires more history features than representing optimal policies. Consider Task 1 for example. Only two binary history features are required to determine the optimal actions: "*is there a cup in front*?" and "*is the second cup on the right-hand side or left-hand side*?". But a perfect Q-function requires more features such as "*how many cups have been picked up so far*?" and "*how far is the second cup from here*?". A perfect model for this task requires the same features as the perfect Q-function. But a perfect model for Task 2 requires even more features such as "*what is the current state of the random number generator*?", while a perfect Q-function for Task 2 requires no extra features.

  It is important to note that we do not need a perfect Q-function or a perfect model in order to obtain an optimal policy. A Q-function just needs to assign a value to each

action in response to a given situation such that their relative values are in the right order, and a model just needs to provide sufficient features for constructing a good Q-function.

6.1 Architecture Characteristics

Given the above problem parameters, we would like to understand which of the three architectures is best suited to which types of problems. Here we consider the key advantages and disadvantages of each architecture, along with the problem parameters which influence the importance of these characteristics.

- **Recurrent-model architecture.** The key difference between this architecture and the recurrent-Q architecture is that its learning of history features is driven by learning an action model rather than the Q-function. One strength of this approach is that the agent can obtain better training data for the action model than it can for the Q-function, making this learning more reliable and efficient. In particular, training examples of the action model (<sensation, action, next-sensation, payoff> quadruples) are directly observable with each step the agent takes in its environment. In contrast, training examples of the Q-function (<sensation, action, utility> triples) are not directly observable since the agent must estimate the training utility values based on its own changing approximation to the true Q-function.

  The second strength of this approach is that the learned features are dependent on the environment and independent of the reward function (even though the action model may be trained to predict rewards as well as sensations), and therefore can be reused if the agent has several different reward functions, or goals, to learn to achieve.
- **Recurrent-Q architecture.** While this architecture suffers the relative disadvantage that it must learn from indirectly observable training examples, it has the offsetting advantage that it need only learn those history features that are *relevant* to the control problem. The history features needed to represent the optimal action model are a superset of those needed to represent the optimal Q-function. This is easily seen by noticing that the optimal control action can in principle be computed from the action model (by using look ahead search). Thus, in cases where only a few features are necessary for predicting utilities but many are needed to predict completely the next state, the number of history features that must be learned by the recurrent-Q architecture can be much smaller than the number needed by the recurrent-model architecture.
- **Window-Q architecture.** The primary advantage of this architecture is that it does not have to learn the state representation recursively (as do the other two recurrent network architectures). Recurrent networks typically take much longer to train than non-recurrent networks. This advantage is offset by the disadvantage that the history information it can use are limited to those features directly observable in its fixed window which captures only a bounded history. In contrast, the two recurrent network approaches can in principle represent history features that depend on sensations that are arbitrarily deep in the agent's history.

Given these competing advantages for the three architectures, one would imagine that each will be the preferred architecture for different types of problems:

- One would expect the advantage of the window-Q architecture to be greatest in tasks where the memory depths are the smallest (for example, the pole balancing task).
- One would expect the recurrent-model architecture's advantage of directly available training examples to be most important in tasks for which the payoff delay is the longest (for example, the pole balancing task). It is in these situations that the indirect estimation of training Q-values is most problematic for the recurrent-Q architecture.
- One would expect the advantage of the recurrent-Q architecture — that it need only learn those features relevant to control — to be most pronounced in tasks where the ratio between relevant and irrelevant history features is the lowest (for example, the cup collection task with two random features). Although the recurrent-model architecture can acquire the optimal policy as long as just the relevant features are learned, the drive to learning the irrelevant features may cause problems. First of all, representing the irrelevant features may use up many of the limited context units at the sacrifice of learning good relevant features. Secondly, as we have seen in the experiments, the recurrent-model architecture is also subject to instability due to changing representation of the history features— a change which improves the model is also likely to deteriorate the Q-function, which then needs to be re-learned.

We do not expect the window-Q architecture to work very well in general, because it can fail to disambiguate world states even when its window size N is chosen large enough to represent the optimal policy. Consider the cup collection task. After picking up the first cup, the agent walks back and forth for M steps ($M > N$). (Note that this kind of random walks often occur during early learning.) The window-Q agent will not know which world state it is currently in, because knowing that requires more history information than that available in the window. (If the window-Q agent follows the optimal path, the bounded history suffices to identify every world state along the path.) In contrast, a well-trained recurrent model can keep track of state transitions and be able to know the actual world state under the same situation.

The tapped delay line scheme, which the window-Q architecture uses, has been widely applied to speech recognition [Waibel, 1989] and turned out to be quite a useful technique. However, as mentioned above, we do not expect it to work as well for control tasks as it does for speech recognition, because of an important difference between these tasks. A major task of speech recognition is to find the temporal structure that already exists in a given sequence of speech phonemes. While learning to control, the agent must look for the temporal structure generated by its own actions. If the actions are generated randomly as it is often the case during early learning, it is unlikely to find sensible temporal structures within the action sequence so as to improve its action selection policy.

7 Related Work

As mentioned in Section 2, there are two basic approaches to addressing the problem of perceptual aliasing: sensor approaches and memory approaches. [Whitehead & Ballard, 1991] and [Tan, 1991] are examples of sensor approaches. Both of the sensor approaches assume the existence of sensory operations by which the actual world state can be unambiguously identified without the need to look back in the past. They also assume deterministic environments and lookup table representations for the Q-function.

In recent years, the multilayer neural network and the recurrent network have emerged and become important components for controlling nonlinear systems with or without hidden states (perceptual aliasing). Figure 9 illustrates several control architectures, which can be found in the literature. Some of these architectures use recurrent networks, and some use just feed-forward networks. In principle, all these architectures can be modified to control systems with hidden states by introducing time-delay networks and/or recurrent networks into the architectures.

Note that there are two types of critic in Figure 9: *Q-type* and *V-type*. The Q-type critic is an evaluation function of state-action pairs, and therefore is equivalent to the Q-function except that the critic may have multiple outputs; for example, one output for discounted cumulative pain and one output for discounted cumulative pleasure. (The Q-nets in Figure 1 can also be modified to have multiple outputs.) The V-type critic is an evaluation function of only states. Temporal difference (TD) methods are often employed to learn both types of critic.

The *adaptive heuristic critic* (AHC) architecture (Figure 9.a) was first proposed by [Sutton, 1984] and later studied by several researchers (e.g., [Anderson, 1987] [Lin, 1992]). Each time an action is taken, the action is rewarded or punished based on whether it leads to better or worse results, as measured by the critic. At the same time, the critic is trained using TD methods. This architecture assumes discrete actions.

The *back-propagated adaptive critic* (BAC) architecture (Figure 9.b) was proposed by [Werbos, 1987] [Werbos, 1990]. This architecture assumes the availability of the desired utility at any time. For example, the desired utility for the pole balancing system is 0 all the time; that is, the pole never falls. Under this assumption and the assumption that the critic and the action model have been learned already, the control policy can be trained by back-propagating the difference between the desired utility and actual critic output through the critic to the model and then to the policy network, as if the three networks formed one large feed-forward network. Here the gradients obtained in the policy network indicate how to change the policy so as to maximize the utility. Note that this architecture can handle continuous actions.

The architecture shown in Figure 9.c was described by [Werbos, 1988] and [Schmidhuber, 1991]. This architecture assumes the availability of the desired outputs from the system to be controlled. Under this assumption, the errors between actual outputs and desired outputs can be back-propagated through the model to the policy network. The gradients obtained in the policy network indicate how to change the policy so as to obtain the desired outputs from the system. Note that this architecture can handle continuous actions.

[Jordan & Jacobs, 1990] proposed a control architecture (Figure 9.d) (also known as 2-net

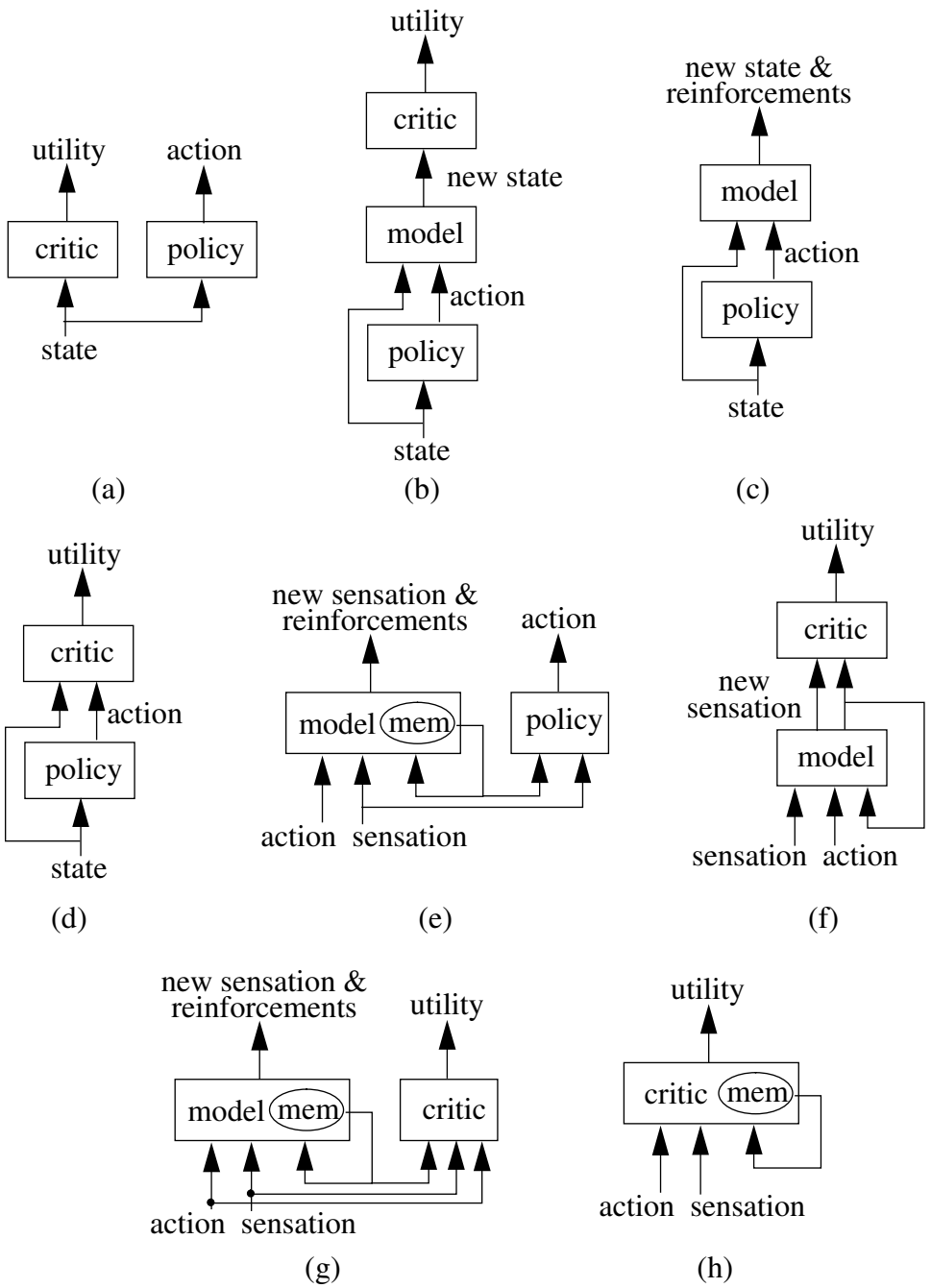

Figure 9: Control architectures.

architecture) somewhat similar to the BAC architecture; instead of using a model and a V-type critic, it uses a Q-type critic. This architecture also assumes the availability of the desired utility and can handle continuous actions.

The architecture shown in Figure 9.e was described by [Thrun and Möller, 1992]. Given a well-trained model, this architecture performs multiple-step look-ahead planning to find out the best action for a given situation. Using the input situation and the action that is found best as a training pattern, the policy network is then trained to mimic the multiple-step planning process in one step. This architecture can handle continuous actions.

The architecture shown in Figure 9.f was proposed by [Bachrach, 1992]. Assuming that the action model has been learned already, this architecture trains a V-type critic using TD methods. The control policy is simply to choose the action for which the critic output is maximal. This architecture assumes discrete actions.

Our proposed recurrent-model architecture, which is shown in Figure 1.c and replicated in Figure 9.g, is similar to Bachrach's architecture (Figure 9.f). Both assume discrete actions and do not require the availability of the desired utility. Two differences between both are that: (1) our architecture uses a Q-type critic, while his a V-type critic, and (2) the Q-type critic uses the actual, current sensation as inputs, while the V-type critic uses the predicted, next sensation as inputs. Because correct predictions by the V-type critic strongly relies on correct predictions by the model, we expect our recurrent-model architecture to outperform Bachrach's architecture, if the learning agent has rich sensations and is unable to learn a good action model.

Our recurrent-model architecture is also similar to a control architecture proposed by [Chrisman, 1992]. A main difference is that our model learning is based on gradient descent and his based on maximum likelihood estimation.

All of the architectures discussed in this section so far consist of two or more components. The recurrent-Q architecture (Figure 1.b and Figure 9.h), on the other hand, consists of only one component, a critic. The critic is directly used to choose actions, assuming that actions are enumerable and the number of actions is finite.

8 Conclusion

This paper presents three memory-based architectures for reinforcement learning in non-Markovian domains: the window-Q, recurrent-Q, and recurrent-model architectures. These architectures are all based on the idea of using history information to discriminate situations that are indistinguishable from immediate sensations. As we have shown, these architectures are all capable of learning some non-Markovian tasks. They are also able to handle irrelevant features and small control errors to some degree. We have discussed strengths and weaknesses of these architectures in solving tasks with various characteristics.

The (good) performance reported in this paper would not be obtainable without two techniques: experience replay with large λ values and back-propagation through time. The following is our general summary on each of these architectures:

1. Surprisingly, the recurrent-Q architecture seems to be much more promising than we thought before this study; we in fact did not expect this architecture to work at all. As long as the memory depth and payoff delay required by a task are not too large, this architecture appears to work effectively.

2. For tasks where a small window size N is sufficient to remove perceptual aliasing, the window-Q architecture should work well. However, it is unable to represent an optimal control policy if the memory depth of the problem is greater than N.

3. The recurrent-model approach seems quite costly to apply, because model learning often takes a lot more effort than what seems to be necessary. The difficulty of the general problem of model learning is well recognized. There are few methods that are truly successful. For example, the *diversity-based inference procedure* proposed by [Rivest & Schapire, 1987] is restricted to deterministic, discrete domains. Instance-based approaches, such as [Moore, 1990], are suitable for nondeterministic, continuous domains, but cannot learn history features. [Chrisman, 1992] studies a model learning method based on a statistical test. That method can handle nondeterminism and history features, but does not seem to scale well. We also do not think the architectures proposed here scale well to problems with very large memory depths. On the other hand, once a model is learned for a task, it may be re-usable for other tasks.

Finally, as mentioned before, combinations of these architectures are possible and may give better performance than the basic versions. Further study of this remains to be done.

Acknowledgements

We thank Chris Atkeson, Ronald Williams, Rich Sutton, and Sebastian Thrun for fruitful discussions on issues related to this work. We also thank Lonnie Chrisman and Sebastian Thrun for helpful comments on a draft of this paper. This research was supported in part by Fujitsu Laboratories Ltd and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

A The Experience Replay Algorithm

Figure 10 shows the experience replay algorithm, which we used to train Q-functions in this work. First we define two terms. An *experience* is a quadruple, (x_t, a_t, x_{t+1}, r_t), meaning that at time t action a_t in response to state x_t results in the next state x_{t+1} and reinforcement r_t. A *lesson* is a sequence of experiences starting from an initial state x_0 to a final state x_n where the goal is achieved or given up. By experience replay, the agent remembers a lesson and repeatedly presents the experiences in the lesson in chronologically backward order to the algorithm depicted in Figure 10, where γ, $0 \leq \gamma < 1$, is the discount factor and λ, $0 \leq \lambda \leq 1$, is the recency parameter used in TD(λ) methods [Sutton, 1988].

To replay $\{(x_0, a_0, x_1, r_0)...(x_n, a_n, x_{n+1}, r_n)\}$, do

1. $t \leftarrow n$
2. $e_t \leftarrow Q(x_t, a_t)$
3. $u_{t+1} \leftarrow Max\{Q(x_{t+1}, k) \mid k \in actions\}$
4. $e'_t \leftarrow r_t + \gamma[(1-\lambda)u_{t+1} + \lambda e'_{t+1}]$
5. Adjust the network implementing $Q(x_t, a_t)$ by back-propagating the error $e'_t - e_t$ through it
6. if $t = 0$ exit; else $t \leftarrow t - 1$; go to 2

Figure 10: The experience replay algorithm

The idea behind this algorithm is as follows. Roughly speaking, the expected return (called TD(λ) return) from (x_t, a_t) can be written recursively [Watkins, 1989] as

$$R(x_t, a_t) = r_t + \gamma[(1-\lambda)U(x_{t+1}) + \lambda R(x_{t+1}, a_{t+1})] \tag{3}$$

where

$$U(x_t) = Max\{Q(x_t, k) \mid k \in actions\} \tag{4}$$

Note that the term in the brackets [ ] of Equation 3, which is the expected return from time *t+1*, is a weighted sum of 1) return predicted by the current Q-function and 2) return actually received in the replayed lesson. With $\lambda = 1$ and backward replay, $R(x_t, a_t)$ is exactly the discounted cumulative reinforcements from time t to the end of the lesson. In the beginning of learning, the Q-function is usually far from correct, so the actual return is often a better estimate of the expected return than that predicted by the Q-function. But when the Q-function becomes more and more accurate, the Q-function provides a better prediction. In particular, when the replayed lesson is far back in the past and involves bad choices of actions, the actual return (in that lesson) would be smaller than what can be received using the current policy (In this case, it is better to use $\lambda = 0$).

Ideally, we want $Q(x_t, a_t) = R(x_t, a_t)$ to hold true. The difference between the two sides of "=" is the error between *temporally successive predictions* about the expected return from (x_t, a_t). To reduce this error, the networks which implement the Q-function are adjusted using the back-propagation algorithm (Step 5). When $t = n$, λ must be set to 0, because e'_{t+1} in Step 4 is undefined. For further discussions, see [Lin, 1991].

The experience replay algorithm can be used in batch mode or in incremental mode. In incremental mode the Q-nets are adjusted after replaying each experience, while in batch mode the Q-nets are adjusted only after replaying a whole lesson. The advantage of backward replay is greatest when the algorithm is used in incremental mode. In this work, non-recurrent Q-nets were trained in incremental mode, while recurrent Q-nets in batch mode to avoid instability.

Table 2: Parameter Settings

	Task 1	Task 2	Task 3	Task 4
window-Q	$N = 5$ $\lambda = 0.8$ $H_q = 0$ $\eta_q = 0.02$ $1/T = 10 \rightarrow 30$	same as Task 1	same as Task 1 except $\eta_q = 0.01$	$N = 1$ $\lambda = 0.7$ $H_q = 6$ $\eta_q = 0.25$ $1/T = 20 \rightarrow 50$
recurrent-Q	$\lambda = 0.8$ $C_q = 8$ $H_q = 14$ $\eta_q = 0.04$ $1/T = 5 \rightarrow 30$	same as Task 1	$\lambda = 0.8$ $C_q = 12$ $H_q = 20$ $\eta_q = 0.02$ $1/T = 5 \rightarrow 30$	$\lambda = 0.7$ $C_q = 2$ $H_q = 10$ $\eta_q = 0.02$ $1/T = 20 \rightarrow 50$
recurrent-model	$\lambda = 0.8$ $C_m = 16$ $H_m = 24$ $\eta_m = 0.02$ $H_q = 10$ $\eta_q = 0.2$ $1/T = 5 \rightarrow 30$	$\lambda = 0.8$ $C_m = 18$ $H_m = 26$ $\eta_m = 0.02$ $H_q = 11$ $\eta_q = 0.2$ $1/T = 5 \rightarrow 30$	$\lambda = 0.8$ $C_m = 20$ $H_m = 30$ $\eta_m = 0.02$ $H_q = 12$ $\eta_q = 0.1$ $1/T = 5 \rightarrow 30$	$\lambda = 0.7$ $C_m = 4$ $H_m = 6$ $\eta_m = 0.005$ $H_q = 6$ $\eta_q = 0.5$ $1/T = 15 \rightarrow 50$

B The Parameter Settings For The Experiments

The parameters used in the experiments included:

- discount factor γ (fixed to be 0.9),
- recency factor λ,
- range of the random initial weights of networks (fixed to be 0.5),
- momentum factor (fixed to be 0 for perceptrons and 0.9 for multilayer networks),
- window size (N),
- learning rate for Q-nets (η_q),
- learning rate for action models (η_m),
- number of hidden units for Q-nets (H_q),
- number of hidden units for action models (H_m),
- number of context units for Q-nets (C_q),
- number of context units for action models (C_m),
- temperature for controlling randomness of action selection (T), and
- P_l (see below).

Table 2 shows the parameter settings used to generate the mean performance described in this paper. (Note that $H_q = 0$ means that we used perceptrons for Q-nets.) Those parameter values were empirically chosen to give roughly the best performance.

The experience replay algorithm did not replay all experience in the past. It replayed only the experience from the most recent 70 trials, and replayed only the actions whose probabilities

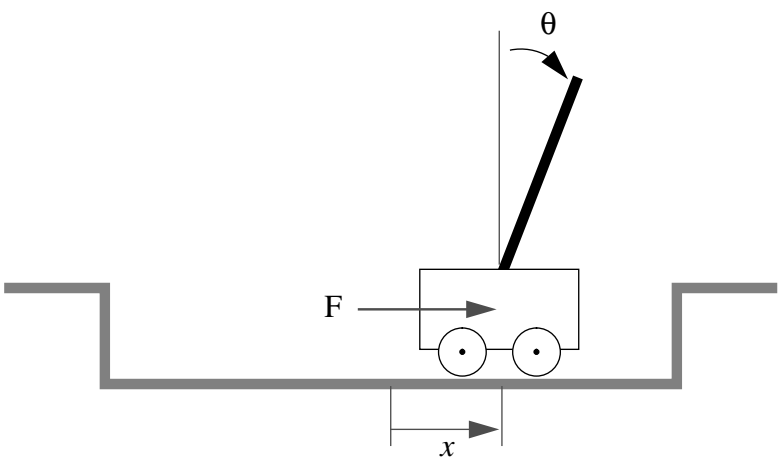

Figure 11: The pole balancing problem.

of being chosen by a stochastic action selector were greater than P_l. (See [Lin, 1991] [Lin, 1992] for discussions.) The value we used for P_l was between 0.01 and 0.00001 for the cup collection tasks and between 0.0001 and 0.0000001 for the pole balancing task, depending on the recency of the experience to be replayed.

C The Pole Balancing Problem

The dynamics of the cart-pole system (Figure 11) are given by the following equations of motion [Sutton, 1984] [Anderson, 1987]:

$$
\begin{aligned}
x(t+1) &= x(t) + \tau \dot{x}(t) \\
\dot{x}(t+1) &= \dot{x}(t) + \tau \ddot{x}(t) \\
\theta(t+1) &= \theta(t) + \tau \dot{\theta}(t) \\
\dot{\theta}(t+1) &= \dot{\theta}(t) + \tau \ddot{\theta}(t) \\
\ddot{x}(t) &= \frac{F(t) + m_p l \left[\dot{\theta}^2(t) \sin\theta(t) - \ddot{\theta}(t) \cos\theta(t)\right] - \mu_c sgn(\dot{x}(t))}{m_c + m_p} \\
\ddot{\theta}(t) &= \frac{g \sin\theta(t) + \cos\theta(t) \left[\frac{-F(t) - m_p l \dot{\theta}^2(t) \sin\theta(t) + \mu_c sgn(\dot{x}(t))}{m_c + m_p}\right] - \frac{\mu_p \dot{\theta}(t)}{m_p l}}{l \left[\frac{4}{3} - \frac{m_p \cos^2\theta(t)}{m_c + m_p}\right]}
\end{aligned}
$$

where

g	=	$9.8m/s^2$	=	acceleration due to gravity,
m_c	=	1.0 kg	=	mass of cart,
m_p	=	0.1 kg	=	mass of pole,
l	=	0.5 m	=	half pole length,
μ_c	=	0.0005	=	coefficient of friction of cart on track,
μ_p	=	0.000002	=	coefficient of friction of pole on cart,
$F(t)$	=	$\pm$10.0 newtons	=	force applied to cart's center of mass at time t,
τ	=	0.02 s	=	time in seconds corresponding to one simulation step,

$$sgn(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0. \end{cases}$$

The reinforcement function is defined as:

$$r(t) = \begin{cases} -1 & \text{if } |\theta(t)| > 0.21 \text{ radian or } |x(t)| > 2.4 \text{ m}, \\ 0 & \text{otherwise.} \end{cases}$$

References

[Anderson, 1987] Anderson, C.W. Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning,* pages 103-114.

[Bachrach, 1992] Bachrach, J.R. *Connectionist modeling and control of finite state environments.* Ph.D. Thesis, Department of Computer and Information Sciences, University of Massachusetts.

[Barto *et al.*, 1991] Barto, A.G., Bradtke, S.J., and Singh, S.P. *Real-time Learning and Control using Asynchronous Dynamic Programming.* Technical Report 91-57, Computer Science Department, University of Massachusetts.

[Chrisman, 1992] Chrisman, L. Reinforcement learning with perceptual aliasing: The predictive distinctions approach. To appear in AAAI-92.

[Elman, 1990] Elman, J.L. Finding structure in time. *Cognitive Science* 14, 179-211.

[Jordan & Jacobs, 1990] Jordan, M.I. & Jacobs, R.A. Learning to control an unstable system with forward modeling. In D.S. Touretzky (ed.), *Advances in Neural Information Processing Systems 2*, pages 324-331, Morgan Kaufmann.

[Lin, 1991] Lin, Long-Ji. Programming robots using reinforcement learning and teaching. In *Proceedings of AAAI-91*, pages 781-786.

[Lin, 1992] Lin, Long-Ji. Self-improving reactive agents based on reinforcement learning, planning and teaching. In *Machine Learning* (in press).

[Moore, 1990] Moore, A.W. *Efficient memory-based learning for robot control.* Ph.D. Thesis; Technical Report No. 209, Computer Laboratory, University of Cambridge.

[Mozer & Bachrach, 1991] Mozer, M.C and Bachrach, J.R. SLUG: A connectionist architecture for inferring the structure of finite-state environments. In *Machine Learning*, 7, 139-160.

[Rivest & Schapire, 1987] Rivest, R.L. and Schapire, R.E. Diversity-based inference of finite automata. In *Proceedings of the Twenty-Eighth Annual Symposium on Foundations of Computer Science*, pages 78-87.

[Rumelhart *et al.*, 1986] Rumelhart, D.E., Hinton, G.E., and Williams, R.J. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol.1,* Bradford Books/MIT press.

[Schmidhuber, 1991] Schmidhuber, J. Reinforcement learning in Markovian and non-Markovian environments. In D.S. Touretzky (ed.), *Advances in Neural Information Processing Systems 3*, pages 500-506, Morgan Kaufmann.

[Sutton, 1984] Sutton, R.S. *Temporal credit assignment in reinforcement learning.* PhD Thesis, Department of Computer and Information Science, University of Massachusetts.

[Sutton, 1988] Sutton, R.S. Learning to predict by the methods of temporal differences. In *Machine Learning*, 3, 9-44.

[Tan, 1991] Tan, Ming. Learning a cost-sensitive internal representation for reinforcement learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 358-362.

[Thrun and Möller, 1992] Sebastian B. Thrun and Knut Möller. Active Exploration in Dynamic Environments. In D.S. Touretzky (ed.), *Advances in Neural Information Processing Systems 4*, Morgan Kaufmann.

[Waibel, 1989] Waibel, A. Modular construction of time-delay neural networks for speech recognition. *Neural Computation* 1, 39-46.

[Watkins, 1989] Watkins, C.J.C.H. *Learning from Delayed Rewards.* PhD Thesis, King's College, Cambridge.

[Werbos, 1987] Werbos, P.J. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics, 17*, 7-20.

[Werbos, 1988] Werbos, P.J. Generalization of back propagation with applications to a recurrent gas market model. *Neural Networks*, 1: 339-356.

[Werbos, 1990] Werbos, P.J. A menu of designs for reinforcement learning over time. In Miller, W.T., Sutton, R.S., and Werbos, P.J., editors, *Neural Networks for Control.* The MIT Press, Cambridge, MA.

[Whitehead & Ballard, 1991] Whitehead, S.D. and Ballard, D.H. Learning to perceive and act by trial and error. In *Machine Learning*, 7, 45-83.