ONLINE STATISTICAL MODELING IN REINFORCEMENT LEARNING

by

JULIAN ANDREW HOOKER, B.S.E.P.

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Chairperson of the Committee

Accepted

Dean of the Graduate School

May, 2004

ACKNOWLEDGMENTS

There are a great deal of people who deserve thanks for me get far enough along to create this document. I will keep the list down to the truly important.

First and foremost, I would like to thank my family for the support over the years of my life. Mom, Dad, and little Amy have gotten me through a lot.

I have collected many friends over the years. In some way, each of them have helped me keep going. Specifically, I would like to thank Bill Ferguson, Heath Garner, Guy Kraemer, Gerald Pippin, Todd Quasny, Joey Severino, and Cohen Steed.

Finally, I would like to thanks Dr. Larry Pyeatt for helping me out throughout the thesis process.

TABLE OF CONTENTS

ABSTRACT

Simulation against a model can greatly improve the learning rate of Reinforcement Learning. The Dyna algorithm uses both real experience and model learning to facilitate simulation. However, the model used in Dyna is fairly limited, yet still has some desirable properties. Examination of a few different known models can help bring to light ways of improving the Dyna model. Combining ideas from what is learned about these models should allow to a greatly improved model for Reinforcement Learning simulation.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

AI (Artificial Intelligence) encompasses a wide variety of subfields (Russell and Norvig 1995). One technique, known as Reinforcement Learning (RL), evaluates reinforcements signals from the environment to learn how to accomplish its tasks. The reinforcements are used to evaluate how good or bad it is to be in a particular situation (state). RL keeps track of the expected reinforcement of each action in each situation and uses this information to perform actions to reach its goal.

There are some limitations involving RL. RL requires large amounts of time to train. RL only learns in small increments and many increments need to be taken before a RL agent can find an optimal strategy for a task. Fortunately, this training issue can be assisted by simulation against a model of the real environment. A model representation of an environment can be used to run the RL agent through the task numerous times in a small amount of execution time. Additionally, complications and damage from a physical world can be avoided with simulation. A RL agent controlling a robot does not need to physically fall down a set of stairs one hundred times to learn that it is a bad thing.

Simulation has its own set of problems as well. Simulating against a model is only useful if an accurate model is used. Models can be as abstract or exact as necessary. Constructing an exact model may be difficult and is not always necessary.

Many types of general models exist. One such model is a Markov Model.

Markov Models are used in Markov Decision Processes (MDPs) and are constrained by the Markov Property. They do not store information about the past and assume all the information needed to make a decision can be obtained from current information(Ronald). Much of Reinforcement Leaning theory also assumes the Markov Property (Singh, Jaakkola, and Jordan 1994). This is another limitation of RL as many environments are not Markovian (Singh, Jaakkola, and Jordan 1994). This limitation can also be compensated for by the use of other types of models.

Hidden Markov Models (HMMs) are similar to Markov Models. However, they separate state information from what is observed from the environment. This additional information allows HMMs to compensate for non-Markovian environments and accurately represent them (Rabiner 1989). This increased power unfortunately comes at computational cost. HMMs allow for a more robust model representation, however they are difficult to build online and computationally intensive to train.

Separation of the environment from the agent is not always desirable. Strictly simulating against a model can lead to inaccurate knowledge of the task dynamics. Some algorithms allow for the learning agent to learn from both real experience from the environment and simulated experience from the model at the same time. Dyna (Sutton 1990) is such an algorithm and can be used to significantly speed up the learning rate of a RL agent.

The goal of this work is to find a middle ground between the basic Dyna model, an MDP, and an HMM. The Dyna model, being deterministic, is too limited. Adding the ability to automatically train stochastic actions in the dynamic building of models should improve upon the Dyna model. In addition, the Hybrid model will

work online similar to the Dyna model by not adding large amounts of computational complexity. This work will show a comparison against the non-model case, the basic model case, and the hybrid model case.

CHAPTER 2

MARKOV DECISION PROCESSES

A Markov Decision Process (MDP) is a process following the Markov Property. The Markov Property states that historical information about what has happened in a process up to the current states is not needed to perform optimal actions or make optimal decisions. Information does not need to be retained from previous states to be able to make a decision in the current state. Mathematically speaking, if we have a process which is described as

$$Prs_{t+1} = s', r_{t+1} = r|s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0 \tag{2.1}$$

becomes

$$Prs_{t+1} = s', r_{t+1} = r|s_t, a_t \tag{2.2}$$

An MDP is defined by the following: a set of states, a set of actions, a set of transition probabilities ($\mathcal{P}^a_{ss'}$), and a set of expected rewards ($\mathcal{R}^a_{ss'}$). The notation $\mathcal{P}^a_{ss'}$ means the probability of reaching state s' given s is the current state and action a was the action taken.

Actions from states have a percentage chance of making it to a particular state and another percentage chance to reach another state. These probabilities introduce an element of randomness to the actions taken in the MDP. This makes

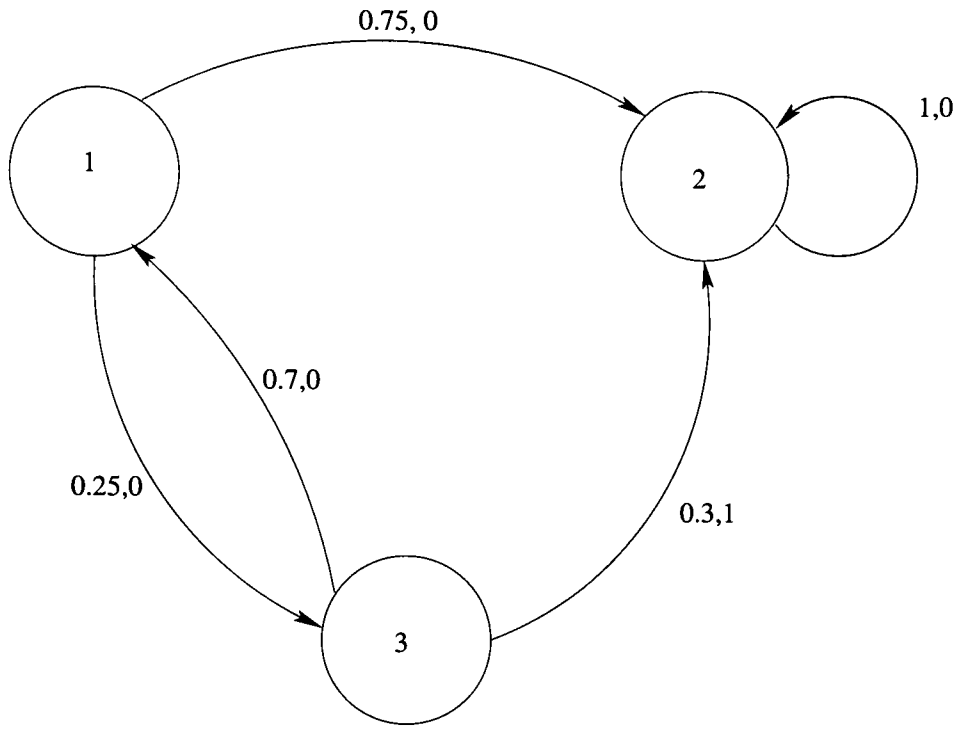

Figure 2.1: A Simple Markov Decision Process

MDPs stochastic.

The simple MDP in Figure 2.1 has three states and one action it can take. In state one the action has a 75% chance of getting to state two, but there is a 25% chance the agent will end up in state three, and so on. Also, if the agent is in state three and performs the action, the agent receives a reward of 1 if it reaches state two. It receives a reward of 0 if it reaches state one.

CHAPTER 3

HIDDEN MARKOV MODELS

Hidden Markov Models (HMMs) extend the idea of the models in an MDP. HMMs abstract away the states from what is observed. The idea of observations is introduced. These output observations correspond to the output of the environment being modeled. Since observations are the output of the models, the actual state space is hidden within the model. HMMs are defined by several things:

Set of output observations. $O = o_1, o_2, ..., o_M$.

Set of states $\Omega = 1, 2, 3, ..., N$.

A transition probability matrix $A = a_{ij}$.

An output probability matrix $B = b_i(k)$.

An initial state distribution $\pi = \pi_i$.

a_{ij}, $b_i(k)$, and π are probabilities and must sum to one. A compact form of the specification is conveniently used based on the fact that 3 probabilities (A,B,π) are needed to define an HMM:

$$\lambda = (A, B, \pi). \tag{3.1}$$

There are three problems regarding HMMs (Rabiner 1989) but the problem of interest in this work is the learning problem. the learning problem deals with training

the HMM based on a set of observation sequences to evaluate the model parameters. Training of the HMM can be done with the Baum-Welch algorithm.

The Baum-Welch algorithm (also known as the forward-backward algorithm) is an iterative algorithm similar to the EM algorithm (Huang, Acero, and Hon 2001) and guarantees an improvement on each iteration. Eventually, the Baum-Welch algorithm will produce a local maximum (Baum and Sell 1968). According to the EM algorithm the maximization process is equivalent to maximizing the following equation (Huang, Acero, and Hon 2001):

$$Q(\lambda, \hat{\lambda}) = \sum_{allS} \frac{P(X, S|\lambda)}{P(X|\lambda)} \log P(X, S|\hat{\lambda}) \tag{3.2}$$

where $P(X, S|\lambda)$ can be expressed as

$$P(X, S|\lambda) = \prod_{t=1}^{T} a_{s_{t-1}s_t} b_{s_t}(X_t) \tag{3.3}$$

and $\log P(X, S|\hat{\lambda}$ can be expressed as

$$\log P(X, S|\lambda) = \sum_{t=1}^{T} \log a_{s_{t-1}s_t} + \sum_{t=1}^{T} \log b_{s_t}(X_t) \tag{3.4}$$

Equation 3.2 can then be written as

$$Q(\lambda, \hat{a}_i) = Q_{a_i}(\lambda, \hat{a}_i) + Q_{b_j}(\lambda, \hat{b}_j) \tag{3.5}$$

where

$$Q_{a_i}(\lambda, \hat{a}_i) = \sum_i \sum_j \sum_t \frac{P(X, s_{t-1} = i, s_t = j|\lambda)}{P(X|\lambda)} \log \hat{a}_{ij} \tag{3.6}$$

$$Q_{b_j}(\lambda, \hat{b}_j) = \sum_i \sum_j \sum_{t \in X_t = o_k} \frac{P(X, s_t = j|\lambda)}{P(X, \lambda)} \log \hat{b}_j(k). \tag{3.7}$$

Knowing all the probabilities must sum to one and these terms have the form

$$F(x) = \sum_i y_i \log x_i \tag{3.8}$$

where

$$\sum_i x_i = 1. \tag{3.9}$$

According to Lagrange multipliers, Equation 3.2 will achieve the maximum value at

$$x_i = \frac{y_i}{\sum_i y_i}. \tag{3.10}$$

And Equation 3.11 and Equation 3.12 follow.

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T} \gamma_t(i, j)}{\sum_{t=1}^{T} \sum_{k=1}^{N} \gamma_t(i, k)} \tag{3.11}$$

$$\hat{b}_j(k) = \frac{\sum_{t \in X_t = o_k} \sum_i \gamma(i, j)}{\sum_{t=1}^{T} \sum_i \gamma_t(i, j)} \tag{3.12}$$

Initialization, choose an initial estimate $\hat{\lambda}$

Repeat until convergence
Set $\lambda = \hat{\lambda}$
E-Step: Compute auxiliary function $Q(\lambda, \hat{\lambda})$ based on λ
M-Step: Compute $\hat{\lambda}$ according to the re-estimation equation to maximize the auxiliary Q-function

Figure 3.1: The Baum-Welch Algorithm

These equations are used to train the model with each new set of observation sequences. The are obviously very computationally intensive with the multiple summations per term. This makes the the model hard to train in real time in an algorithm like Dyna (chapter 5). Additionally, the structure of the model is not built in real time. It needs to be known before the start of the simulation.

The algorithm for Baum-Welch is in Figure 3.1.

CHAPTER 4

REINFORCEMENT LEARNING

When first learning to walk, children learn via feedback from their experiences. They learn taking several steps to their parents waiting arms is a good experience (hugs, affection, candy), while falling is a bad experience (pain, injury, death). RL agents learn the same way. They take actions in their environment and evaluate that action based on feedback.

4.1 Returns

Evaluation of the action can be a complicated task. Generally evaluation is based on a goal state or set of states set by the agent within a task. For each step in the task the agent evaluates each bit of feedback from the environment. Single step feedback is called a reward (r). As the task progresses, the agent collects a series of rewards.

$$r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T = R_T. \tag{4.1}$$

R_T is the true return on the series of rewards after step t and ending at step T. Feedback received from the environment a long time ago should not have as great a weight as feedback received the last time step. A discounting rate solves this problem. An example of this can be given by,

$$R_T = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+h+1}. \tag{4.2}$$

where γ is known as the discounting rate and is $0 \leq \gamma \leq 1$.

There are two types of tasks: continuous and episodic. Continuous tasks continue with no known end, such as a control system to a valve which is to hold a certain value. Episodic tasks have a distinct start state and end state or states. The child's task of learning to walk starts when the child stands up and ends when the child falls or reaches its parents. A single notation can be used for both types of tasks by introducing an absorbing state in the episodic case. The absorbing state transitions only to itself and always returns a reward of zero. Now, return becomes

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{4.3}$$

Returns are only feedback for the agent. They do not tell the agent how to behave or what actions to take. The tool used to control the behavior of the agent is a value function. Value functions keep track of states in the environment and values for those states based on rewards previously seen in those states and the value of states around them. Also, the value function knows the state transition matrix for those states.

4.2 Value Functions

The value function can be used to determine what action to take in a state based on the expected return for that action. This mapping of state to action is called

the agent's policy (π). This mapping of can be seen as

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} \tag{4.4}$$

and is known as state-value function which follows policy π. Value functions can also be represented as action-value functions. They are similar to state-value functions but take into consideration both the state and the action in the value function

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} \tag{4.5}$$

where $Q^{\pi}(s,a)$ is known as a action-value function which follows policy π.

Taking into consideration the current policy, transition probabilities for actions in this state, any rewards based on those transition probabilities, and the current value of the states that could be transitioned to from this state, a value function, V, following policy, π, is

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'}[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')] \tag{4.6}$$

where $\pi(s,a)$ is the policy value for taking action a in state s, $\mathcal{P}^{a}_{ss'}$ is the transition probability from state s to state s' when taking action a, $\mathcal{R}^{a}_{ss'}$ is the reward for taking action a in state s and ending up in state s', γ is the discounting factor and $V(s')$ is the value of state s'. This state-value function (Equation 4.6) is known as the Bellman equation.

Hopefully, an agent's policy will lead the agent to its goal. Ideally, the policy

will maximize its expected return. A policy which maximizes expected return is an optimal policy (π^*). Since value functions map an agent's policy, an optimal policy can be found from the optimal state-value function

$$V^*(s) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} V^*(s')] \tag{4.7}$$

or the optimal state-action function

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')]. \tag{4.8}$$

An optimal policy is a greedy policy. It always follows whatever actions produce the greatest rewards. However, this is not always desirable. When learning a policy, RL agents can take random actions. This encourages exploration of the environment and helps the agent break out of local maxima. An agent which follows its policy most of the time, but chooses random actions some percentage of the time, is said to be ϵ-greedy.

4.3 Value Iteration

Value iteration is the process of evaluating estimate state values based on returns. An untrained RL agent can use value iteration to calculate values in a value function. It does this through training. The agent executes a task many times until values in the value function converge to the correct values. The algorithm for value iteration is shown in Figure 4.1. Each iteration performs a one step backup of values

Initialize V arbitrarily, e.g., $V(s) = 0$, for all $s \in S$

Repeat
    $\delta \leftarrow 0$
    For each $s \in S$
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$
        $\delta \leftarrow \max(\delta, |v - V(s)|)$
until $\delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
    $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$

Figure 4.1: Value Iteration

in the value function based on the current state (s), states that can be transitioned to from the current state (s'), and the actions to get to those states (a).

Value iteration can be either online or offline. Online value iteration updates the value function it is following, while offline value iteration updates a value function it is not using as its current policy.

Temporal difference (TD) methods are commonly used for value iteration. Two examples of TD methods are SARSA and Q-learning.

4.4 SARSA

SARSA is an online temporal difference method. SARSA learns an action-value pair as opposed to a state-value. SARSA gets its name from the quintuple of events which it uses (s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})It evaluates values for action-value pairs based on values of state-action pairs available from the current state, backing up the

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode)
    Initialize s
    Choose a from s using policy derived from Q (e.g., ϵ–greedy)
    Repeat (for each step in the episode)
        Take action a, observe reward r, and next state s'
        Choose a' from s' using policy derived from Q (e.g., ϵ–greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$, $a \leftarrow a'$
    Until s is terminal

Figure 4.2: SARSA(Sutton and Barto 1998)

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode)
    Initialize s
    Repeat (for each step in the episode)
        Choose a from s using policy derived from Q (e.g., ϵ–greedy)
        Take action a, observe reward r, and next state s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$
    Until s is terminal

Figure 4.3: Q-Learning (Sutton and Barto 1998)

value of future state-action pairs to the current state-action. SARSA is guaranteed to produce an optimal policy. Figure 4.2 is the SARSA algorithm.

4.5 Q-leaning

Q-learning is an off-policy temporal difference method. Q-learning also uses state-action values ($Q(s,a)$) and is guaranteed to find Q^* regardless of the policy being followed. The algorithm for Q-learning is shown in Figure 4.3.

CHAPTER 5

DYNA

There are four major parts to the Dyna algorithm: evaluation (value function), policy, environment, and the model. The value function is the reinforcement learning value function discussed earlier. Policy is the set of actions taken based on the value function. The environment is the task the agent is trying to solve. The model is an internal representation of the environment.

Dyna allows a Reinforcement Learning agent to learn from both real experience and simulate from a model in the same way. Value iteration uses both the real experience and simulation to find a policy.

Dyna first allows the agent to evaluate new value function values based on real experience with the environment. Then, Dyna adds the experience to a model which represents direct experience with the world. This model is then simulated against, using value iteration to produce a more and more accurate value function representing the environment as seen to that point. Each simulation step simulates random occurrences in the model. The algorithm continues to gather real experience, add it to the model to increase its accuracy, and then simulate the agent against the model to increase the accuracy of the value function. Policies evaluated from value function converge to optimal much more quickly, especially in environments which change slowly. The Dyna algorithm is shown in Figure 5.1.

As stated above, Dyna dynamically builds its model based on real experience.

Initialize $Q(s,a)$ and $Model(s,a)$ *for all* $s \in \mathcal{A}(s)$
Do forever:
(a) $s \leftarrow$ current(nonterminal) state
(b) $a \leftarrow \epsilon$-greedy(s,Q)
(c) Execute action a; observer resultant state, s' and reward, r
(d) $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma max_{a'} Q(s',a') = Q(s,a)\right]$
(e) $Model(s,a) \leftarrow s', r$ *(assuming deterministic environment)*
(f) Repeat N *times:*
$s \leftarrow$ random previously observed state
$s \leftarrow$ random action previously taken in s
$s', r \leftarrow Model(s,a)$
$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma max_{a'} Q(s',a') = Q(s,a)\right]$

Figure 5.1: The Dyna Architecture (Sutton and Barto 1998)

However this model is limited by the fact that the model needs to be deterministic (Sutton and Barto 1998). A deterministic model means that an action in state s will always transition to state s'. There is no randomness in the transitions of a deterministic model. This makes the Dyna model less robust then other stochastic (or non-deterministic) models such as Markov Models (Chapter 2) and Hidden Markov Models (Chapter 3) which can model random transitions within the environment.

CHAPTER 6

SETUP

The setup used is a close adaptation of the Dyna architecture. An object oriented framework is used to produce high reuse of code and allow another layer of abstraction for the various pieces. A trial using the framework only needs 3 pieces: the environment, the model, and the agent. The framework works in such a way so each piece can be used and substituted without need to change or substitute the other pieces.

The three classes (environment, model, agent) have a set of methods which must be implemented. These methods are defined by the framework and ultimately used by it to run each trial.

The environment class implements the task. The environment is self contained and only interacts with any of the other classes through a read-only method. The environment keeps track of everything internally, including the current state of the trial. Outside factors are not allowed to change the environment. There is only one exception, the reset method is used each time a new trial runs.

The agent only takes input and returns actions. Internally, the agent is responsible for evaluating the value function and returning actions based on values in the value function. The agent implementation is open to the implementer.

The model is a representation of the environment. It can take new information as input to add to the current model. It should not have a notion of a current state,

```
Environment:
    takeAction(Action, State) returns (Reward)

Agent:
    getAction(State) returns (Action)
    update(InitialState, Action, FinalState, Reward)

Model:
    add(InitialState, Action, FinalState, Reward)
    getRandomState() returns (State)
    getRandomAction(State) return (Action)
    takeAction(InitialState, Action, FinalState) return (Reward)

All:
    reset() returns to the initial conditions
```

Figure 6.1: Framework API

because it should be able to approximate the environment at any random point in the simulation (Chapter 5). Since implementation of the model is done in a separate class, it can be as exact or approximate to the environment as needed. The framework API is shown in Figure 6.1.

One function (main) calls all of these different pieces through calls to these methods. Other methods and classes can exist, but at least these methods must be publicly exposed.

6.1 Mountain Car

Mountain car is a classic problem in reinforcement learning. A car at the bottom of a valley is to get to the top of one of the hills around it. Unfortunately, at the maximum velocity the car can achieve, gravity still prevents the car from making

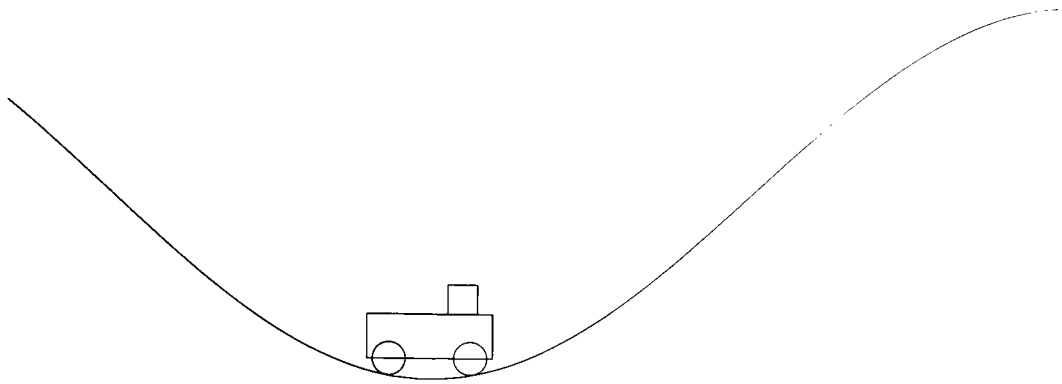

Figure 6.2: Mountain Car

it to the top of the hill (Figure 6.2). Therefore, the car must learn to back up the opposite side of the hill and accelerate down that side to increase its forward velocity and get to the top of the hill on the other side. This is a relatively simple environment to start with because only two parameters are needed to learn the task, the position and velocity of the car.

Three sets of trials are run. The first is just the agent and the environment, no simulation takes place. Second, the agent is simulated against the Dyna model in addition to the environment. Finally, the Hybrid model is used, adding stochastic events to the modeling.

The learning agent uses Q-Learning to do its value iteration. The RL agent uses an α of 0.1, discounting factor (γ) of .85, and chooses random actions 20% of the time. However, starting at episode 20, the randomness decreases .05% every 5 episodes. This gives the learning agent the chance to explore in the beginning, yet still follow a greedy policy after significant learning has taken place.

Both of the models have similar characteristics. Each discretizes their state space into a 30 by 30 grid. A large grid is needed to overcome problems of losing

```
Method from framework:
    takeAction(InitialState, Action) return (FinalState, Reward)

if InitialState previously seen
then
    if Action previously seen in InitialState
    then
        replace previous Action, FinalState and Reward in
        InitialState with new values
    else
        add Action, FinalState, Reward to InitialState
else
    add InitialState, Action, FinalState, Reward to model
```

Figure 6.3: Dyna Method Add

determinism when discretizing the state space. This is a problem for the Dyna model (Chapter 5). When in use, both models are used for 5 simulation steps. Each model also designates the individual states to record which actions can be take from that state and what state is the result of that action. This is where the models differ.

The Dyna model keeps track of only deterministic transitions. For each initial state, it stores what action is being taken in that state and where that action transitioned the first time it sees a new action for that state. Each time that action is seen again in that initial state, the previous values are overwritten by the new values. The algorithm the Dyna method uses for adding states to the model is in Figure 6.3

The Hybrid model stores the initial state, any action taken in the initial state, and stores multiple state transitions from the actions in the initial state. Any time an action is seen in an state, the model records where that action transitions in a probability distribution function (pdf) for that action in that state. When state transitions are requested in a state during simulation, this pdf is used to determine

```
Method from framework:
    takeAction(InitialState, Action) return (FinalState, Reward)

if InitialState previously seen
then
    if Action previously seen in InitialState
    then
        add FinalState, Reward to pdf for Action
    else
        add Action, FinalState, Reward to InitialState
else
    add InitialState, Action, FinalState, Reward to model
```

Figure 6.4: Hybrid Method Add

what final state that action reaches. These are stochastic transitions. This algorithm appears in Figure 6.4

Each trial is run 50 times and averaged over 30 runs. A graph of the number of episodes each method took per trial appears in Figure 6.5. Initially, all of the trials perform many steps per episode. Because of this, the resolution in Figure 6.5 after the learning of the task is hard to see. Figure 6.6 is much easier to see. The first two episodes are left off of the graph intentionally. For each method the number of steps is the same on the first episode, since nothing is yet known about the environment. The second episode is still very large for the direct RL case and skews the scale on the graph. This graph demonstrates how each method converges to optimal and how they compare to the convergence of the other methods. As can be seen in Figure 6.6, both models find the optimal policy more quickly, finding the optimal policy withing a few episodes. The direct RL agent takes around 30 episodes to find the optimal policy. The Dyna model and the Hybrid model both seem to have similar characteristics in

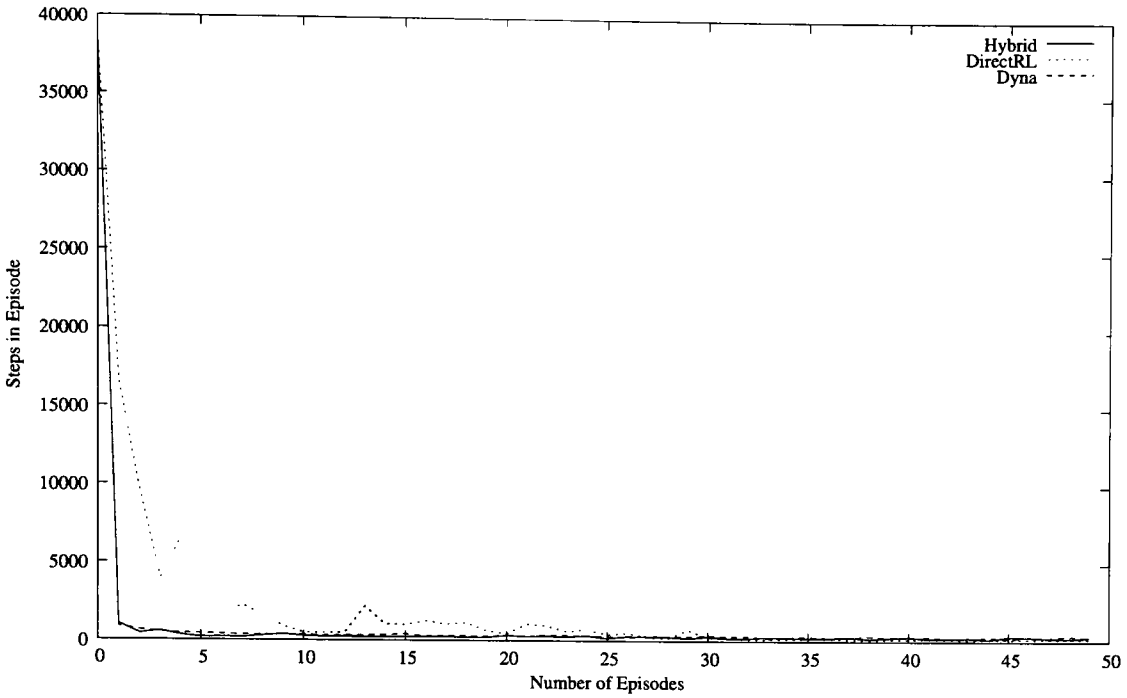

Figure 6.5: Mountain Car Number of steps per episode complete

this graph.

Figure 6.7 shows the how long each episode takes to run. This graph is also taken from 50 episodes averaged of 30 runs. The direct RL agent runs in a much shorter amount of time, as to be expected. Again, both of the model lines are similar. This suggests that both models are about the same in performance.

Looking at the graphs suggests some similarities in both the Dyna model and the Hybrid model. To determine how similar to each other they are, a Student's t-test can be run. The Student's t-test can be used to determine if data sets are statistically different or not. For the tests the data sets are broken up into 2 regions. The two regions are the region the agents are learning and then the region after the learning has finished. The results are shown in Table 6.1.

A "Similar" in the table means the two methods are statistically similar. "Dif-

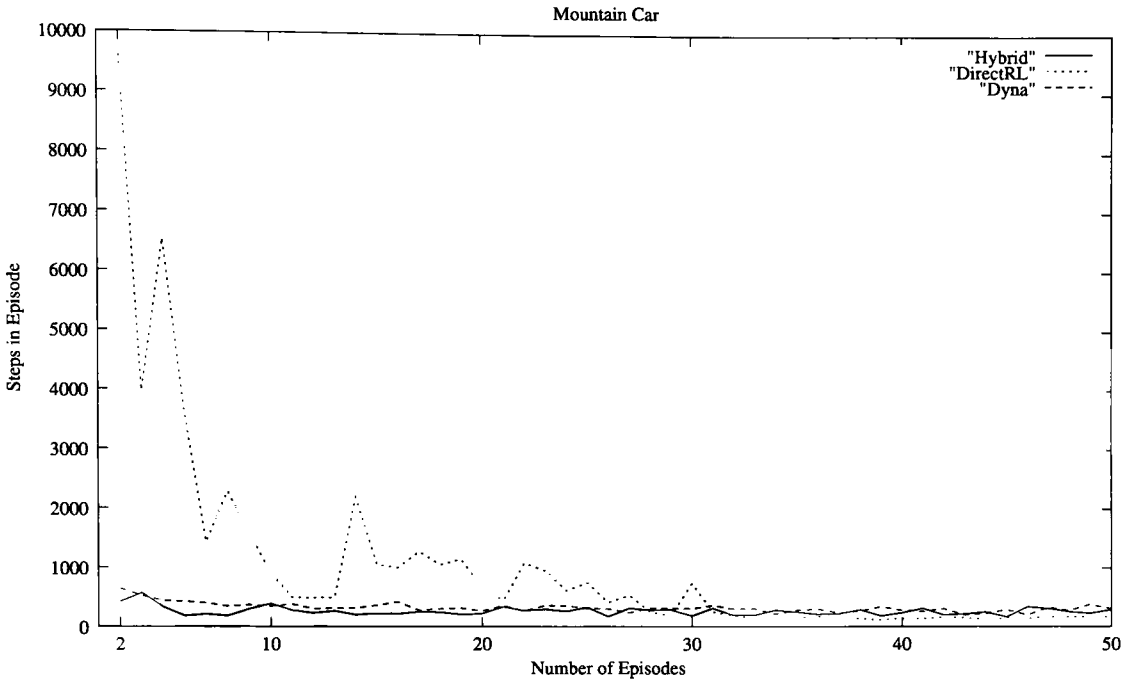

Figure 6.6: Mountain Car - Number of steps per episode modified

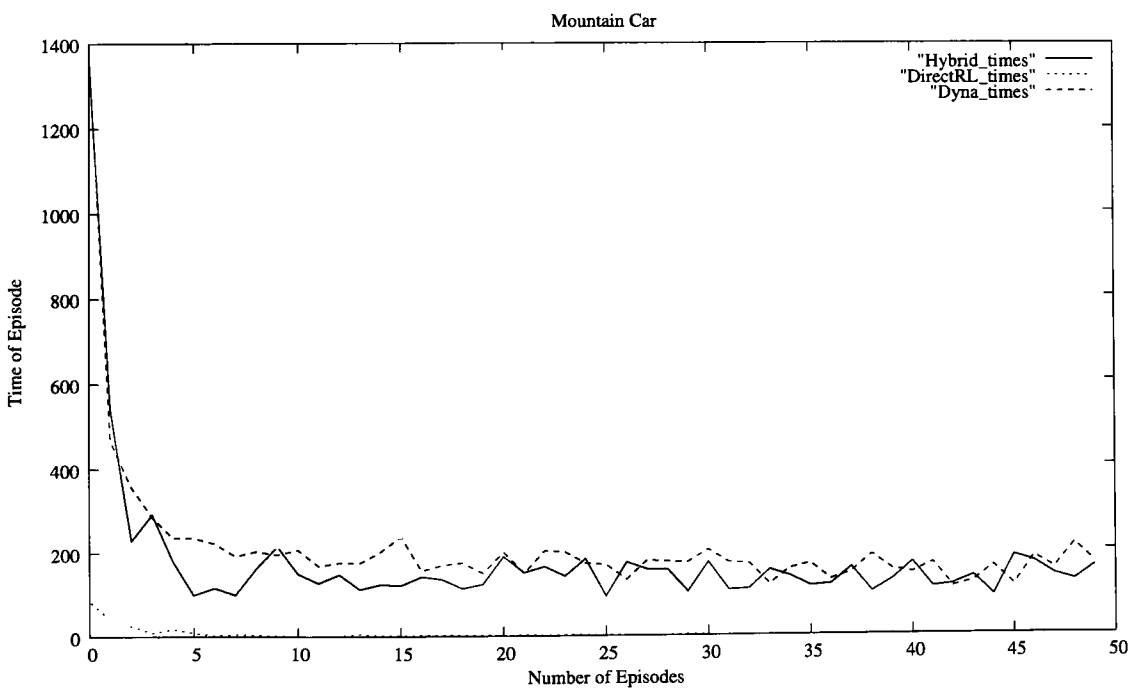

Figure 6.7: Mountain Car Time per episode

Table 6.1: Method Comparison

Methods	Learning	Post-learning
Direct Dyna	Different	Similar
Direct - Hybrid	Different	Similar
Dyna - Hybrid	Similar	Different

ferent" means the methods are different. It can be seen that the models are statistically similar, however different than the direct RL method. However, once the optimal solution has been found, the model learning becomes similar to the direct RL method. Although, they do differ slightly from each other. The agent using Dyna had a little trouble with the optimal policy, probably due to the non-deterministic nature of the of the state space.

CHAPTER 7

CONCLUSIONS

Figures 6.6 and 6.7 and additionally table 6.1 show proof that this work has realized its goals. The Hybrid model can handle stochastic transitions in an online environment. It performs comparable to the Dyna model without adding a lot of additional CPU time. Also, once the optimal policy is found, the direct RL combined with the Hybrid model produce similar results, as to be expected. However, the direct RL and Hybrid model combination greatly increase the learning rate, similar to the direct RL and Dyna model combination.

CHAPTER 8

FUTURE WORK

While the Hybrid model demonstrated similar performance and modeling to the Dyna model, there were are few assumptions it knew beforehand. The maximum and minimum of each parameter was known before the start of the trials. This information could easily be determined during the course of the trials. However, the maximum and minimum of each parameter affects the size of the states in the model. As the maximum and minimum of a parameter changes, the size of the state would grow or shrink. It would be interesting if these changes in size affect how well the agent can learn using the model or if some scaling of the states and the values in them would be needed for each change.

Also, the Hybrid model presented in the work learns an environment. This environment's behavior is assumed to be the same throughout the trials. A study could be made to see if the Hybrid model could adapt to changes in the way the environment behaves. An example might be a robot learning an obstacle course. The robot's actions might be stochastic through the course, and the course is modeled by the Hybrid model. However, once the course is learned, what happens if it changes? This type of study could extend ideas found in this work.

At the beginning of the learning process, the model is not at all usable on its own. However, the hybrid model produced by the end of the learning process is actually a MDP representing the environment. Currently this MDP is lost at the

end of the trials. Future effort could be put into retrieving this MDP to be usable in future simulations.

Finally, the environment used for testing is fairly simple to learn. Mountain Car only needs two parameters in order to learn an optimal policy. Testing the hybrid model method against more complicated environments (those requiring more parameters to learn) and checking the results would be necessary.

Baum, L. E. and G. R. Sell (1968). Growth functions for transformations on manifolds. *Pac. Journal of Math 27*, 211–227.

Huang, X., A. Acero, and H.-W. Hon (2001). *Spoken Language Processing.* Englewood Cliffs, NJ: Prentice Hall.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speach recognition. In *Proceedings of the IEEE, Vol. 77, No. 2*, pp. 257–287.

Ronald, B. Hierarchical control and learning for markov decision processes.

Russell, S. and P. Norvig (1995). *Artificial Intelligence A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall.

Singh, S. P., T. Jaakkola, and M. I. Jordan (1994). Learning without state-estimation in partially observable markovian decision processes. In *International Conference on Machine Learning*, pp. 284–292.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 216–224.

Sutton, R. S. and A. G. Barto (1998). *Reinforcement Learning: An Introduction.* Cambridge, Massachusetts: MIT Press.

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

_____ _____

Student Signature Date

Disagree (Permission is not granted.)

_____ _____

Student Signature Date