University of Alberta

Library Release Form

Name of Author: Adam Murray White

Title of Thesis: A Standard Benchmarking System for Reinforcement Learning

Degree: Master of Science

Year this Degree Granted: 2006

Adam Murray White
9999 111 st
Edmonton, Alberta
Canada, T5K 1K3

Date: _____

Nothing in life is to be feared. It is only to be understood.

– Marie Curie.

University of Alberta

A STANDARD BENCHMARKING SYSTEM FOR REINFORCEMENT LEARNING

by

Adam Murray White

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

in

Department of Computing Science

Edmonton, Alberta
Fall 2006

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Standard Benchmarking System for Reinforcement Learning** submitted by Adam Murray White in partial fulfillment of the requirements for the degree of **Master of Science** in .

_____

Richard S. Sutton

_____

Dale Schuurmans

_____

Marek Reformat

Date: _____

To Patrick,
you have been a father, a big brother and my best friend

Abstract

We introduce a standard framework for benchmarking in reinforcement learning. Benchmarks facilitate the comparison of alternative algorithms and can greatly accelerate research progress. The University of California Irvine (UCI) machine learning database, for example, was very effective for driving progress in supervised learning. Creating a similar benchmarking resource for reinforcement learning is more challenging because reinforcement learning agents and environments interact to generate observations, actions and rewards. The observations and rewards received by the learning agent depend on the actions; these training data cannot simply be stored in a file as they are in supervised learning. Instead, the reinforcement learning agent and environment must be interacting programs. Our benchmarking framework is a standard for communication between these programs. Our protocol 1) guarantees exact reproducibility of the execution sequence of a learning experiment, 2) enables plug and play interchanging of environments and agents, 3) is general and powerful yet non-intrusive, 4) is easy to convert existing agents and environments to. The current implementation features a light-weight software design with layered functionality, support for multiple programming languages and support for agent and environment interaction across a network. We illustrate these advantages with examples from the newly established University of Alberta Reinforcement Learning Library, which is based on our protocol. We conclude by presenting benchmarks for the Grid-world, General Cat and Mouse, Schapire Cat and Mouse, Blackjack, Sensor Network, GARNET, Acrobot, Random, Delayed, Stochastic, and Non-stationary Mountain Car tasks.

Acknowledgements

There are a number of people that have helped me throughout my Masters and with my thesis. First and foremost, I would like to thank my supervisor Rich Sutton. Rich has helped me develop a desire to strive for only the best. I could never have imagined that I would have the opportunity to work with such a world class researcher and a great human being. Thank you Rich. Mark Ring has been a great mentor and helped me believe in myself. He has always been there to talk about new ideas and given me great advice. I would also like to thank my fellow lab mates for all their support and wisdom, specifically: Brian Tanner, Cosmin Paduraru, Anna Koop, Andrew Butcher and Mark Lee.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Benchmarks allow researchers to compare alternative approaches and can help focus and thus accelerate research progress in a field. A benchmark assigns a quantitative score to the performance of an algorithm or method. This scoring allows researchers to rank methods against one another and determine each method's relative strengths and weakness in a variety of situations. A ranking, produced by a benchmarking system, can establish a baseline for theoretical advances. The performance distinction provided by benchmarking also allows researchers to select suitable algorithms for real-world applications, making a field more attractive to industry.

In reinforcement learning, researchers are often forced to report results on environments that have been specifically designed for their algorithms, and researchers must provide comparisons to implementations of algorithms found in the literature. Until now, comparing results from different publications and recreating results from the literature has been problematic because there are few standard software implementations of classic test problems; there is disagreement on the performance metrics that should be collected or the evaluation methodologies that should be used during testing. Researchers who wish to compare their new algorithm(s) to existing results must often re-implement everything. Comparisons of this kind can be inaccurate and often not empirically verifiable.

In supervised learning, however, research progress is guided by the University of California Irvine (UCI) Machine Learning Repository. The UCI repository is the primary database of standardized benchmark problems for supervised learning. The UCI repository contains thousands of data sets in a free online database. The UCI database makes it

possible for researchers to evaluate new learning algorithms on the same data sets used to evaluate algorithms in the literature–UCI guarantees exact reproducibility of results. We describe the UCI machine learning repository in more detail in Section 2.3.

The success of the UCI repository is difficult to replicate in reinforcement learning because reinforcement learning agents and environments interact to generate observations, actions and rewards. The agent selects actions based on its current observation of the environment and the reward. The environment's state transitions are governed by the action selected by the agent. Each unique experience trajectory corresponds to a unique sequence of these agent-environment transactions. It is not possible to store the environmental data responses for all possible state action combinations; the dynamics of the environment cannot be encoded in a static data set as they are in supervised learning.

In reinforcement learning, the learning agent and environment must be interacting programs. The agent and environment program must exchange actions, observations and rewards, in sequence, thousands of times over the course of a learning experiment. The communication between the agent and environment must be standardized so that a benchmark system can be established; a communication protocol can facilitate such standardization. If the agent and environment communicate according to a standard protocol, we can exactly reproduce the execution that occurs during a learning experiment. Result reproducibility is the main objective of a benchmarking system for reinforcement learning.

Our benchmarking framework is a standard for communication between the agent and environment programs. We introduce our benchmarking framework RL-Glue: a communication protocol for agent and environment interaction based on a function interface. We also introduce the University of Alberta Reinforcement Learning Library (RL-Library): a library of code based on our protocol. We illustrate how the current implementation of our protocol directly addresses many of the challenges to standardizing empirical analysis in reinforcement learning with several benchmarks on examples from RL-Library.

Chapter 2

Benchmarking in Other Fields

Benchmarking has accelerated research progress in computer hardware design, numerical analysis and supervised learning. The hardware community was one of the first to establish a standard benchmarking system to rank advances in hardware architecture designs. The supervised learning community later established a database of test domains and a standard interface for benchmarking regressions and classification algorithms. Hardware and supervised learning benchmarks demonstrate how important benchmarking is to advancing the state-of-the-art in a field and also highlight the variety in benchmarking methodologies.

2.1 Hardware Benchmarks

Hardware benchmarks were established to provide researchers with a quantitative ranking of different computer chip designs and hardware architectures. It is almost impossible to identify the value of a new data bus or a new Ethernet switch without a benchmark performance number. To benchmark a circuit design, memory scheme, cluster communication architecture or a supercomputer, one must run a benchmark program from an online database maintained by organizations such as Standards Performance Evaluation Corporation (SPEC) [Powers et al., 1995] or National Aeronautics and Space Administration (NASA). These programs produce several numerical benchmarks including CPU time, memory usage, CPU utilization and IO bandwidth, etc. These benchmarks summarize a hardware system's performance relative to any other system that has publicly available statistics on the same benchmark program. The benchmark results are typically distributed and maintained by the distributors of the benchmarking programs. Using this simple inter-

face, researchers can benchmark newly purchased hardware and new developments against machines from around the world.

Hardware benchmarks developed early out of the need for comparison. Hardware researchers must often present performance results on the standard benchmarks to illustrate the contribution of new advances to conference and journal reviewers. Hardware benchmarks provide an excellent example of how effective benchmarking can be, given a simple user interface and publicly available test domains and benchmark results. This universal availability of benchmarks has stimulated rapid development and has become a integral part of the hardware community.

In reinforcement learning, the state-of-the-art has steadily advanced based on empirical comparisons and theoretical convergence and complexity results. However, as the field matures, benchmarks will be needed to evaluate new advances against the multiplicity of competing methods to further research progress and promote more application driven research in reinforcement learning.

2.2 Numerical Analysis

Benchmark based evaluation has also been successfully employed in numerical linear algebra. In numerical analysis, the performance of an algorithm encoded in a program must be ranked against other programs. An algorithm is benchmarked on a number of data sets available online. A benchmark consists of several runtime summary statistics such as mean squared error, number of arithmetic operations and execution time. The performance of the algorithm on the standard data sets forms a benchmark providing a ranking against other methods with publicly available benchmarks on the same data sets. For instance, the efficiency of a linear system solver can be benchmarked against others in the literature by testing several sample problems from Matrix Market [Boisvert et al., 1997].

The goals of the developers of Matrix Market are similar to that of the reinforcement learning community: define the state-of-the-art, characterize industrial-grade applications, promote research through challenge problems, provide a baseline performance for researchers and provide a means to evaluate new algorithmic developments. RL-Glue and RL-Library were designed to address many of the goals for the reinforcement learning com-

munity.

2.3 Supervised Learning

UCI Machine Learning Repository was established in 1995, to address many of the challenges that reinforcement learning faces now. At that time, supervised learning had developed into a mature field with a significant base of algorithms, theories and practical successes. However, like reinforcement learning, many theoretical and practical problems remained largely unsolved. The success of benchmarking in other fields promoted researchers at UCI to develop a database of classification problems that would serve as the standard set of test problems for supervised learning and regression algorithms. Today, the UCI repository contains thousands of data sets that measure an algorithm's learning speed, data efficiency, quality of classifier learned and amount of over fitting [Newman et al., 1998]. The interface to the UCI test sets are simple: learning algorithms are trained on a static set of observation response pairs, then performance on the test set is measured. This simple data-file interface is one of the reasons why the UCI repository has been so successful in standardizing empirical analysis in supervised learning.

The UCI repository has become the main resource for supervised learning test beds and is continually growing due to significant community support. UCI makes it easier to determine what problems supervised learning algorithms should be tested on. The supervised learning conferences and journals expect new advances to illustrate performance on the UCI data sets. The UCI repository consolidated a large number of learning approaches and methodologies into a single collective community and is partially responsible for the renewed interest in supervised learning methods in the last decade.

The data file interface between learning algorithms and test domains simplifies many of the difficulties involved in designing a benchmarking system. The communication protocol between learning agent and problem, selection of standard performance measures and compatibility between different classes of learning algorithms is implicitly standardized by the data-set evaluation methodology. Recreating the success and impact of the UCI repository in other fields can be difficult.

Chapter 3

Benchmarking in Reinforcement Learning

In reinforcement learning, we want to benchmark agent and environment programs that interact over a number of time steps. Benchmarking the performance of programs that alter their behavior at runtime based on the history of interactions introduces a number of standardization challenges. In this chapter, we describe these challenges in detail and explicitly list the goals and ambitions for benchmarking in reinforcement learning. We begin with an introduction to the reinforcement learning problem.

3.1 Reinforcement Learning

The reinforcement learning text contains a thorough explanation of many advanced reinforcement learning topics [Sutton and Barto, 1998]. We only provide a brief introduction to the basic reinforcement learning concepts here to give the reader the necessary background for the work presented in later chapters. Discussion of more advanced reinforcement learning algorithms can be found in the references.

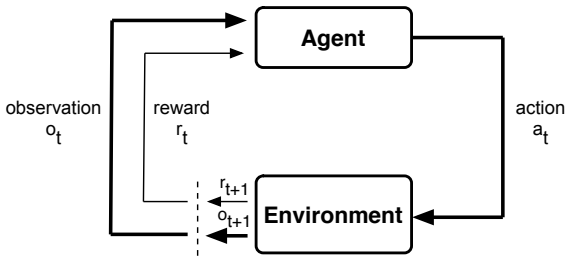

Figure 3.1: The agent-environment interaction in reinforcement learning.

In reinforcement learning the agent learns through interaction with an environment, from the consequences of action, rather than from explicit teaching. The agent interacts with its environment by selecting actions based on its observation of the environment and a reward signal. The reward is a scalar value produced by the environment that provides a numerical score for the agent's behavior. In the reinforcement learning framework (see Figure 3.1), the agent and environment interact continually, exchanging observations, rewards and actions. Each interaction is called a time step. In an episodic task, the sequence of time steps is called an episode.

Consider the Mountain Car task introduced by Moore [1990]. In the Mountain Car domain, an agent must drive an underpowered car up a steep mountain road. The difficulty is that the force of gravity is stronger than the car's engine. Even at full throttle the car cannot accelerate up the slope. The car's movement is described by two continuous observation variables: position and velocity of the car. There is one continuous action, the control variable for the angle of the acceleration pedal. A reward of -1 is assigned on every step, until the car reaches the goal region on the east end of the valley (termination occurs). The agent's objective is to drive out of the valley as fast as possible—maximize reward.

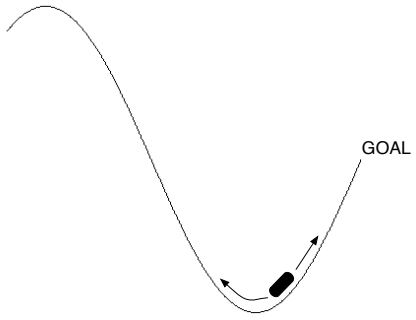

Figure 3.2: The Mountain Car domain.

The temporal sequence of agent-environment interactions that occurs during an episode of the mountain car task can be summarized as follows: on the first time step of an episode the environment positions the car at the bottom of the valley with zero velocity and returns the initial observation (o_0). The agent then selects an action (a_0) based on the initial observation. On the next time step the environment updates the car's position and velocity based on the agent's action and produces a new observation and a reward (o_1, r_1). The agent

then selects a new action (a_1) based on it's new observation (o_1). This execution sequence continues until the car reaches the goal state.

In the Mountain Car Problem, the environment can transition into the goal state producing a terminal observation (o_T). In a continuing task, however, the agent and environment interact forever, i.e., $T = \infty$. The full temporal sequence that occurs during the agent and environment interaction proceeds as follows:

$$o_0,\ a_0,\ r_1,\ o_1,\ a_1,\ \ldots\ ,\ r_{T-1},\ o_{T-1},\ a_{T-1},\ r_T,\ o_T \tag{3.1}$$

The Markov Decision Process (MDP) formalism, on which most reinforcement learning theory is based, assumes the agent can observe the complete state of the environment on every time step. The next state and reward depend only on the current state and reward, independent of the temporal sequence that leads to the current state. In many interesting domains, however, the agent observes partial information about the state of the system. In this case, the agent must estimate the values and select actions based on observations $o \in O$ of the underlying state of the MDP. An observation that is equal to the state of the MDP is a special case of an observation. In this work we consider the more general framework of partially observable MDPs (POMDP) and use the term "state" only when describing the theoretical basis of reinforcement learning and "observation" otherwise.

In the MDP framework, the agent selects actions to maximize its cumulative reward over time. The agent's policy provides a mapping from each state of the MDP, $s \in S$, to an action a from the set of actions available to the agent in state s (denoted $A(s)$). Most reinforcement learning algorithms improve the agent's policy based on an estimate of the value function, which is a mapping from states to the expected discounted sum of future rewards given an initial starting state s and policy π:

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\}$$

where $\gamma \in (0, 1]$ is a discounting factor and t is the current time step. Similarly, the state-action-value function $Q^{\pi}(s, a)$ is the expected sum of future rewards starting in state s, taking action a and then following π thereafter.

Many reinforcement learning methods are based on temporal difference (TD) learning [Sutton and Barto, 1998]. TD methods update the value of each state $V(s)$, as they are

visited based on the estimated value of the next state and the reward (r_{t+1}): updating a guess with a guess. At time t:

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

where α controls the rate of learning. TD learning algorithms can learn the state-action-value function $Q(s, a)$ and update the policy π to solve control tasks, such as Mountain Car, by selecting the action a that maximizes the state-action-value for all $s \in S$.

Some tasks have high-dimensional or continuous-valued states. The algorithms described above are based on table lookup, they store the value of each state in a table indexed by the state label. In the Mountain Car task, a tabular value function cannot be used to store the value of each state because the state of the environment is comprised of two real-valued numbers. A function approximation method handles large or infinite state spaces by generalizing between states that are deemed similar and finely discriminating between states that are unrelated. Many different function approximation methods are used in the literature such as neural networks, radial basis functions, Kanerva coding, state aggregation and statistical methods like Gaussian Processes, see, e.g; [Engel et al., 2005, Sutton and Barto, 1998]. Many reinforcement learning methods, however, use linear tile coding for function approximation. Linear function approximators are easy to implement, the speed of learning scales linearly with the number of features, and the resultant representation is easy to analyze. Tile coding discretizes a continuous input into binary feature vectors by partitioning the state space with multiple overlapping tilings. Using this representation, the TD learning algorithms can be extended to learn the values of approximated states.

The reinforcement learning model of intelligence is simple yet expressive, borrowing heavily from well-established fields such as psychology, cognitive science and neuroscience. Animal learning experiments have shown that small animals such as mice and larger animals with more complex brains can learn to complete various tasks with a simple reward schedule like navigating a maze or pushing a button for food [Hafner, 2005, Niv et al., 2005]. At the neural level, reinforcement signals have been shown to have strong links with intelligent behavior [Montague et al., 1995]. Reinforcement learning algorithms,

such as TD learning, represent one of the prominent models for reward-associated learning processes in the brain [Niv et al., 2005]. More practically, reinforcement learning methods have been successfully applied to many large scale and complex control tasks, such as elevator control [Crites and Barto, 1996], automated dialog system control [Singh et al., 2002], stock trading [Nevmyvaka et al., 2006] and robotic soccer [Stone and Sutton, 2001]. Model based reinforcement learning methods have been successfully used to fly a helicopter through a variety of maneuvers while upside down [Ng et al., 2004]. This feat is even more impressive when we consider that no human operator has been able to replicate some of the maneuvers the agent learned. Finally, the world's strongest backgammon program, TD-gammon [Tesauro, 1995], used reinforcement learning methods to learn how to play the game at a grand-master level. TD-gammon is now considered the best backgammon player in the world. It has not only beaten the best human players, but its moves have produced new strategies that have been adopted by grand-masters due to their tactical superiority.

The work on TD-gammon and inverted helicopter control illustrates how trial and error learning can be used to learn complex polices. However, there are still many open research challenges in reinforcement learning, such as reducing the variance of off-policy methods [Precup et al., 2005], developing new continuous action selection methods [Williams, 1992], speeding convergence of online learning [Engel et al., 2005], developing more efficient exploration methods [Simsek and Barto, 2006] and improving function approximation techniques [Boyan and Moore, 1995, Sutton, 1996]. These problems are all related to one of the open challenges in reinforcement learning research: scaling reinforcement learning methods to solving applications from industry. Could the same theory and algorithms used to solve Mountain Car be extended to controlling water flow in a hydroelectric dam? Establishing a standardized methodology for ranking the relative strengths of different algorithms can help researchers begin to answer these kinds of questions.

3.2 Challenges to Benchmarking in Reinforcement Learning

In reinforcement learning, a communication protocol is needed to query the agent and environment programs and efficiently communicate data between the two. There are a large variety of communication schemes available. Maximizing execution efficiency and repre-

sentational generality in a single architecture involves a number of design decisions and tradeoffs.

The hierarchy of control must be carefully designed to support a wide variety of learning agents. The agent might call the environment to acquire the new observation and reward, alternatively the environment could control the execution of the agent. Perhaps both the agent and environment programs could be connected to unique communication modules that perform a "handshake" operation and exchange data. We could implement a decentralized scheme where neither the agent nor environment program has calling control. Our communication protocol must be designed to facilitate rigid empirical analysis in a general fashion.

The identification of the modules in our system architecture also demands careful consideration. An agent, environment and central control module seem like a natural base on which to build a benchmarking system, but is this sufficient? There are many other components that could be added to the system to specialize its functionally. A "rules" module could be added for game applications. A "coordinator" module could also be used for multi-agent tasks. Perhaps the reward module should be separated from the environment to allow more complex reward schedules. Again, we must take care to balance redundancy with expressive power and generality with specialization.

An agent and environment communication protocol must not only determine execution schedules but also make the data necessary for learning available to the agent program during a learning experiment. The agents observation of the environment may be prohibitively large requiring significant computational resources. For example, it may not be possible to efficiently pass sonar, laser range finder and vision data between the agent and the environment on every step. Alternatively, it may be very costly for the environment to produce new observations. The amount of data communicated between the agent and environment must be minimized while avoiding expensive calls to environment functions. Furthermore, a learning agent might exploit information about the environment's dynamics to accumulate more reward if it could access environment data. Should the agent be allowed to observe the randomness of the environment, the hidden states or the reward function, for example? The communication protocol should provide the agent with all the information necessary to

learn without revealing data that allows the agent to "cheat".

It seems quite natural to expect a learning agent to perform well on a number of related tasks in similar domains. We expect a tennis-playing agent to also play ping-pong or badminton adequately. However, this implicit similarity is not realized in the strict syntax of programming languages. If the dynamics of a task are changed, agent programs should still be capable of learning. The specifics of the environment, such as the observation dimensions and ranges and the number and types of actions, could be communicated to the agent so that one algorithm can be executed on multiple environments without changing source code. The protocol should provide the agent with a problem description without revealing information about how to actually solve the problem. What and when information is passed by a protocol determines the generality of agent programs that can be benchmarked.

Earlier attempts to establish benchmarking systems have faced some opposition because of dependencies on particular programming methodologies and restrictions on agent and environment format. The communication protocol at the heart of a benchmarking system should not be dependent on popular programming paradigms; researchers should not have to learn a new programming language to use a benchmarking system. Furthermore, forcing researchers to implement mandatory interface code makes it difficult to convert existing agents and environments to a new standard. Previous attempts to standardize benchmarking illustrate the difficulties associated with standardizing the implementation language for reinforcement learning. It is difficult to select a single programming language that is ideal for all classes of reinforcement learning agents or environments. A benchmarking system that supports a number of different programming languages is more likely to gain community support.

Ideally, a benchmarking system for reinforcement learning could also be used to standardize evaluation in competitions. In past reinforcement learning competitions, participants enter a number of agent programs which are later evaluated on several test domains. Competitions introduce a number of additional design challenges. A benchmarking system used for competitions must be efficient and also be light-weight so that existing agent software can be easily converted to work with the system. Competitions also must deal with the tradeoff between learning efficency and computational complexity. Is an algorithm

that converges to the optimal policy in 20 episodes better than one that converges in 2000 episodes, if the later runs in one-thousandth the time of the former? The benchmarking system needs to provide performance measurements that allow us to address these questions. Finally, the experiment and environment code must be separated from the agent program to ensure fairness in competition: each learning agent should be tested under the same experimental settings and participants should not be allowed modify the problems to improve performance. In general, we must standardize as much of the evaluation process as possible so that winners can be decided fairly.

Numerous technical issues arise when implementing a communication protocol in software. When a new version of the software is released, what should be done with existing agents and environments? A versioning system must be established so compatibility issues do not retard the development and growth of the benchmarking system. Furthermore, multi-language support introduces a number of software implementation challenges, such as conversion of data types and data structures between languages. How does one convert an observation represented in programming language to a different data type in another language in a general fashion? It is important that these software implementation issues not compromise the design goals of the benchmarking system.

3.3 Goals

There are several goals that motivate the development of a benchmarking system for reinforcement learning. These goals represent a researcher-oriented view of benchmarking. A benchmarking system for reinforcement learning should:

- establish a suite of standard versions of benchmark problems
- allow researchers to replicate results from the literature
- remove the need to re-implement others' code
- set performance baselines for algorithm development
- facilitate accurate comparisons between algorithms on standard benchmarks

These goals can be achieved in a variety of ways. Our system attempts to satisfy a number of more concrete sub goals to achieve the goals listed above. The agent-environment communication protocol has been designed to:

- exactly reproduce the execution sequence of a learning experiment
- provide plug and play interchanging of agents and environments
- support a large variety of reinforcement learning algorithms

The above goals should not be compromised by the software implementation of the benchmark system. Practical implementation issues and software engineering principles must be balanced with the design principles of the communication protocol. Our benchmark system has been implemented to:

- support any implementation language for agent and environment programs
- allow easy conversion of existing agents and environments to the RL-Glue standard
- provide a light-weight interface with minimal mandatory agent and environment code
- be computationally efficient and easy to use

RL-Glue and RL-Library are designed to achieve all these goals. We have developed a benchmarking system and web-based software library that balances simplicity, efficiency, flexibility and expressive power. In the next chapter, we provide a case study of the Mountain Car problem highlighting the standardization and evaluation challenges discussed above.

Chapter 4

Related Work

Establishing a standard communication protocol among agents and environments and standardizing performance measurements has been well studied in reinforcement learning and multi-agent systems. In reinforcement learning, several research efforts have focused on developing a standard agent-environment communication protocol to facilitate the establishment of benchmarks. In this section, we survey three previous efforts to establish a standard interface in reinforcement learning. We discuss each system's approach to the evaluation and standardization issues in reinforcement learning and describe the differences between these implementations and RL-Glue. Much of the design of RL-Glue is built upon the groundwork laid down by these earlier specifications.

4.1 RL-Interface

The specification and implementation of RL-Glue was largely influenced by the long development history of the RL-Interface [Sutton and Santamaria, 1996]. RL-Interface was developed by Richard Sutton in 1995. RL-Interface has gone through several major transitions over the past ten years. The history of RL-Interface provides several useful insights: the benefits of object-oriented programming, the choice of implementation languages and required functionality. We describe each iteration of the RL-Interface briefly, providing an account of how the RL-Interface's development shaped RL-Glue.

The first major version of RL-Interface used object-oriented programming to implement a function-based interface between agents and environments. The object-oriented approach facilitated the establishment of hierarchies of agents and environments. Inheritance was

used to specify various attributes and generic methods of agents and environments common to all reinforcement learning algorithms (for example, the `agent-step` method). The object-oriented framework facilitates easy plug and play of agents and environments without additional compiling and allows the interfaces functionality to be extended with minimal effort (towards the multi-agent reinforcement learning problem, for example). The RL-Interface had implementations in C, C++ and LISP. This version of RL-Interface had much of the functionality of RL-Glue, but did not provide a control module: users needed to rewrite experimental code (experiment program in RL-Glue) for every agent-environment pair. The RL-Interface was a mature piece of software, however, the reinforcement learning community seemed reluctant to embrace a standard benchmark system at the time.

The next major evolution of the RL-Interface, renamed the "Reinforcement Learning Toolkit" (RL Toolkit), employed the Python scripting language to implement an object-oriented and purely functional communication interface. RL Toolkit logically separated the system into agent, environment and control module. Object-oriented programming was made optional to increase code readability and decrease execution time. The functionality of RL Toolkit is similar to RL-Glue, but, lacks several important features. RL Toolkit did not provide any means for specifying trajectories of experience, distinguishing between training and testing phases of evaluation nor implementing general task-independent agents. RL Toolkit also illustrated how difficult it is select a single programming language for empirical evaluation in reinforcement learning. The establishment of a language independent system seems critical to the success of establishing a benchmark standard for the reinforcement learning community.

4.2 CLS^2

Other than RL-Interface, Closed Loop Simulation System (CLS^2) was one of the first publicly distributed interfaces for reinforcement learning. CLS^2 was developed by the Neuroinformatics Group at Osnabrueck in 2003 [Riedmiller et al., 2003]. CLS^2 allows a single controller (agent) and plant (environment) to communicate through a standard set of functions. The environment specifies a set of functions for initialization, shutdown and the beginning and end of an episode. The environment also implements `next_state` and

`get_observed_state` functions. The `next_state` method specifies the dynamics of the environment, while the `get_observed_state` method returns the observation of the environmental state. The agent specifies a similar set of functions including a `get_action` method which specifies the learning rule and action selection policy. A control loop calls the environment and agent in turn, passing the observations and actions back and forth throughout the course of an episode until termination. This architecture separates the system into an agent, an environment and a central controller that calls the agent and environment objects.

The software implementation of CLS^2 allows researchers to test agents on environments written in C++. The user interacts with the system through a configuration text file, specifying various parameters for the learning experiment (for example, number of episodes, statistics to collect and training scheme), agent name, environment name and a number of graphics options. The user also specifies the range of starting states, legal states and termination states (as sets) in the configuration file. The system parses the configuration file, performs a learning experiment and displays a graphical interpretation of the problem.

CLS^2 is similar to RL-Glue in its overall design and approach to standardizing communication between agents and environments: a central module that calls functions specified by the agent and environment controls interaction. This interface guarantees the format of agents and environments and standardizes the communication during a learning experiment. CLS^2 is, in fact, based on RL-Interface.

CLS^2 differs from RL-Glue in several important ways. The user is not allowed to directly interact with or control the central-control module. The user must interact with the system through a configuration file, limiting the system to a finite set of configurations and execution modes defined by the configuration language. In RL-Glue the user interacts with the system through the experiment program, providing more flexibility. CLS^2 employs a configuration file to specify starting (valid starting states of an episode), work, illegal and termination states. This design requires the experimenter to have some knowledge of the environment dynamics (particularly the environmental state) in order to use the system. This design also allows a user to specify termination conditions that do not agree with the environment dynamics, making the system less robust. Finally, the software im-

plementation of CLS^2 forces researchers to write the agent and environment code in C++. Ideally, researchers could test new agents written in one language on a number of environments written in different languages. The architecture of RL-Glue allows agents and environments to be written in any programming language.

4.3 RLBench

RLBench was developed by Langford and Bagnell in 2004. RLBench standardizes communication between agents and environments through an I/O interface. Agents and environments have no standard form; the system only requires that environment processes be given a constant stream of data with a particular format [Langford and Bagnell, 2004]. The agent writes states, random seeds and actions in ASCII to a buffer on each step of an episode. The environment reads the data in, updates its internal state and writes the next observation, reward and state to the buffer. This I/O interface removes the need for a central interface or driver program to specify and conduct a learning experiment. The system standardizes the mechanics of the environment by defining several generative models that dictate the format and interval of data delivery from the agent. In this sense, the control module and experimental setup (configuration file in CLS^2 and experiment program in RL-Glue) have been amalgamated into the agent in RLBench.

RLBench uses several generative models to implement different testing paradigms. The most expressive and specific model is the Deterministic Generative model. In this model, the environment expects a state, random seed and action to be sent across the pipe on each time step. The environment then returns the next observation, reward and state. The Deterministic Generative model is used to generate specific sequences of experience for batch training or model building. For example, the agent can experience the same trajectory through the state space by specifying a starting state and random seed at the beginning of every episode. In the Generative model, the environment expects a state and action and returns the next observation, reward and state on each time step. This model is used to generate specific sequences of experience with no control over the stochasticity of state transition dynamics. Finally, in the Trace model, the environment expects an action and returns the next observation and reward on each time step. The Trace model encapsulates

the reinforcement learning problem in its simplest form. Environments produce next observations and rewards based on the action selected by the agent. This model provides no control over randomness or the visitation of states in the state space.

RLBench introduced several novel functionalities that are now part of RL-Glue. The use of different generative models to specify trajectories of experience inspired the `env_set_state` and `env_set_random_seed` functions in RL-Glue. RLBench's I/O communication scheme also inspired the pipe interface RL-Glue uses to facilitate multi-language support.

However, the I/O interface of RLBench differs from the functional interface specification of RL-Glue in several important ways. RLBench does not provide any mechanism for distinguishing between training and testing phases. RL-Glue supports this two-phase evaluation by signaling the end of training to the agent by a call to the agent's agent_freeze function. The amalgamation of control and experimental setup into the agent in RLBench places more implementation burden on the agent write. The learning agent and experimental setup should be independent and separable. The experiment program in RL-Glue is responsible for experimental setup and presentation of results, removing unnecessary complexity and code redundancy in the agent writing process.

Chapter 5

A Case Study on Standardization: The Mountain Car Problem

Since it was first introduced by Moore in 1990, the Mountain Car control problem has been widely used to evaluate advances in learning agents. Mountain Car is interesting because it highlights the issues caused by real-valued observations. A learning agent must use a function approximator to discretize the state space. Mountain Car is also interesting because a successful control policy must drive the car backwards, up the other side of the valley, to give the car enough momentum to drive forwards up the hill. The learning agent must learn to move the car away from the goal, incurring negative reward, to reach the goal.

The Mountain Car problem nicely highlights the current standardization challenges in reinforcement learning empirical evaluation. There is no standard Mountain Car software and also no consistent problem formulations in the literature. This makes it difficult to compare results across publications or recreate existing results. In this chapter, we survey several Mountain Car software packages and several papers that use Mountain Car for empirical evaluation, highlighting these standardization problems.

5.1 Mountain Car Software

Mountain Car has been referred to as one of the standard test problems for reinforcement learning [Wiewiora et al., 2003, Smart and Kaelbling, 2000]. This is misleading because there is little agreement in the literature on the car dynamics or the reward function. A brief search yielded five software packages that implement different variations of the problem. Let us consider each implementations environmental dynamics, ignoring agent-

environment communication and control code provided with the software.

The environment distributed by Sutton implements the dynamics specified by Sutton and Barto [1998]. This version differs from Moores problem specification in the action space: Sutton discretizes the actions to full reverse, neutral and full forward, while Moore defines the actions as continuous quantities. Furthermore, Sutton employs random start states, -1 reward per time step and a goal state defined as an position greater than the top of the hill with any velocity [Sutton, 2000]. The Mountain Car environment in the CLS^2 software package, by the Neuroinformatics Group at Osnabrueck, implements the same car physics as Sutton. The Osnabrueck implementation, however, uses an initial car position at the bottom of the valley with zero velocity and a reward of 0 on every time step except in the goal region, where the reward in the goal region is a function of the car's velocity [Riedmiller et al., 2003]. The Mountain Car environment by Wingate [2004] implements similar dynamics to Sutton's, but the reward function is 0 on every time step and $+1$ if the car stops at the top of the hill. Furthermore, the car's initial position is the middle of the valley with slightly negative initial velocity [Wingate, 2004]. The Mountain Car software in the Massachusetts Reinforcement Learning Repository is based on the Sutton implementation and is thus identical [Mahadevan, 1997]. Finally, Szepesvari's version of Mountain Car implements the same dynamics as Sutton. However, the car is initially positioned in the middle of the valley with a low magnitude random velocity and the reward is 1 minus a function of the cars velocity if the car slows just below the top of the hill, $+1$ if the car is past the top of the hill, -1 if the car drives off the west end of the valley and zero otherwise [Szepesvari and Smart, 2004].

Aside from the Sutton and Mahadevan versions, all implementations of Mountain Car differ in either the initial state of the environment, the reward function, or the environment dynamics. Furthermore, we only compared the environment code, not the experimental setup or the interface between the agent and the environment. Results produced by the Sutton and Massachusetts implementations can not be fairly compared if the control code performed a different trailing schedule, used different starting states, or did not collect similar performance measures. This variety across implementations raises a number of standardization questions. Which implementation should be considered the standard? How can one

decide what version to use when testing new algorithms? Should each researcher implement their own code based on experimental descriptions in the literature? Unfortunately, the variety of performance measures and problem descriptions found in the literature make empirical comparison of results on Mountain Car even more complicated than the software differences mentioned above.

5.2 Mountain Car Formulations

Previous Mountain Car formulations suffer from the same standardization issues that make comparing different software implementations difficult. The physical equations governing the car's movement are not often presented. Some papers do not fully specify the experimental conditions, such as the number of episodes, averaging scheme or initial and goal states. Some papers reference problem specifications from previous works and these earlier papers reference even older papers. This forces the reader to follow a chain of references for the full problem specification. This kind of chain referencing can cause information loss and confusion over which version of the problem is considered the standard. However, one of the biggest obstacle to comparison of results across publications is the variety of performance measures used in papers. Number of steps, cumulative reward, average reward, episode return, execution time and mesh plots of the approximated value and state-action-value functions are often presented but rarely consistently across papers. This inconsistency gives results generated on different versions, with different performance measures, little scientific weight. Worse, some papers directly compare results achieved on one version of the Mountain Car problem to results presented in previous works. If we have no mechanism for determining the compatibility of these different versions, how can such comparisons be made fairly?

I have surveyed several well-known works that use Mountain Car to benchmark advances in learning techniques: Smart and Kaelbling [2000] on instance based locally weighted regression algorithms, Boyan and Moore [1995] on function approximation, bootstraping and off-policy learning, Wiewiora $et\ al$ [2003] on introducing domain knowledge to classical learning algorithms, Riedmiller's [2005] work on combining Q-learning and Neural network function approximation, Bagnell's [2004] thesis on algorithms for noisy robotic

	citation	physical equations	initial conditions	**Rewards:**	F(velocity) ε goal	+1 ε goal	-1 per step	0 per step	**Actions:**	discrete	continuous	**Evaluation:**	steps to goal	value function plot	gradient plot	other	based on best policy	based on first episode	varying α and λ
1	✓				✓					✓			✓				✓			
2				X					X					✓						
3	✓			X					X				✓							
4				X						✓						✓	✓	✓		
5	✓					✓		✓			✓			✓	✓		✓			
6	✓	✓	✓				✓			✓			✓							✓
7			✓				✓		X				✓	✓						✓

Table 5.1: Papers using Mountain Car test domain: (1) [Smart and Kaelbling, 2000], (2) [Boyan and Moore, 1995], (3) [Wiewiora et al., 2003], (4) [Riedmiller, 2005], (5) [Bagnell, 2004], (6) [Singh and Sutton, 1996]. (7) [Sutton, 1996]. An "X" under a category label indicates that no description of the catagory was provided in the paper.

systems, Singh and Sutton [1996] on the effectiveness of replacing eligibility traces and Sutton's [1996] work on reinforcement learning and linear function approximation. Table 5.1 summarizes the information presented in each of the surveyed papers. I examined six main categories: whether the work references a problem description or experimental setup in previous work, if the physical equations governing the car's movement are specified, if the initial conditions are specified, the reward function, the action type and performance measures used to evaluate learning.

Of the six papers summarized in Table 5.1, only two use and reference Moore's problem formulation. Bagnell's work does not clearly identify whether he was using Moore's version of Mountain Car or a variant used by Sutton [Sutton, 1996]. Four of the papers provide no citation and no formal description of the problem dynamics. Smart and Kaelbling cite the Singh and Sutton paper, which in turn cites Moore's thesis. Two papers do not describe the experimental setting or how the data was collected, while the others provide only brief descriptions. Four papers plot the average number of steps to goal per episodes, whereas Boyan & Moore, Bagnell and Sutton present the learned value function, which is difficult to use in a comparison of performance. Riedmiller presented several metrics based on specific policies, making it difficult to assess the speed of learning. The Sutton paper (7) made direct comparisons to the results presented in Boyan & Moore paper (2). Neither (2 and 7) provided a problem description, referenced a formulation of the problem from the literature or provided a reference to the software implementation of the problem.

Chapter 6

The RL-Glue Protocol

Our communication protocol connects four components (see Figure 6.1): agent, environment, experiment programs and the RL-Glue function interface that connects everything together. The experiment program specifies the experimental settings for a learning experiment and controls the execution of RL-Glue. The RL-Glue interface controls the execution sequence that occurs during an episode; it is responsible for calling the agent and environment programs and for communicating data between the two on every time step. The agent program specifies the learning algorithm and action selection policy, while the environment contains the problem specification, state transition dynamics and the reward function. The execution control is realized through parameterized function calls, as illustrated in Figure 6.1. At the top level RL-Glue provides a library of functions to the experiment program and ultimately the researcher, while at the lower level RL-Glue provides an interface between the agent and environment programs.

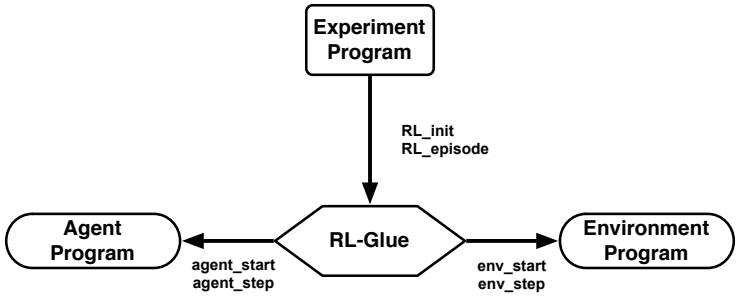

Figure 6.1: The RL-Glue protocol. Arrows indicate function call direction.

RL-Glue standardizes the execution sequence and data transmission that occurs during a learning experiment because every episode produces the same trace of function calls.

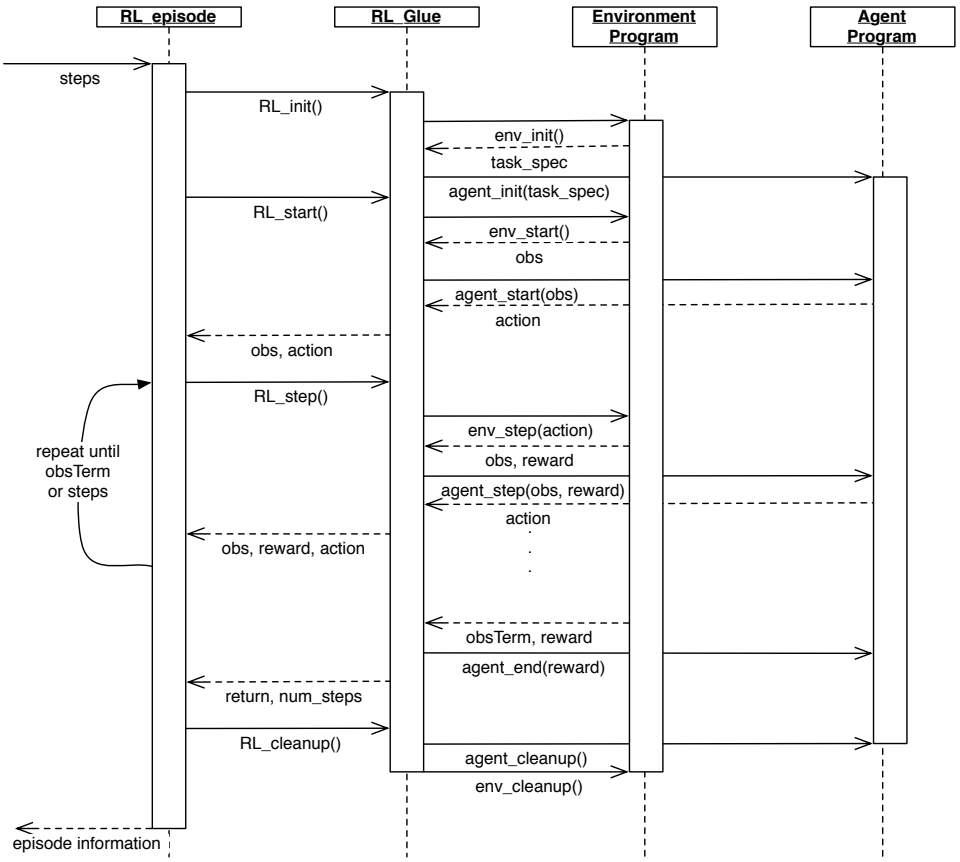

Figure 6.2: The execution sequence that occurs during a call to `RL_episode`.

Consider the execution of a single episode. The experiment program begins by making a call to the RL-Glue episode function to perform a single episode. RL-Glue responds by making calls to the agent and environment to perform the first step of an episode. RL-Glue then calls the agent's and environment's step functions in turn, passing the new reward and observation to the agent the and the new action to the environment. This sequence of calls is continued until the environment indicates to RL-Glue that it has reached a terminal state. RL-Glue responds by calling the agent's end-episode routine and returning control to the benchmark. The sequence of calls performed during an episode is identical (as shown in Figure 6.2) regardless of the agent, environment or type of learning experiment being conducted. In this section, we describe the functions that make up the RL-Glue interface.

6.1 RL-Glue Functions

The functionality of RL-Glue has been layered to make the system light-weight and easy to use for most learning tasks; additional functions are required to achieve more flexibility in experimental design and to implement more complicated agent programs. The RL-Glue functions allow the user to freeze the agent's policy and control the experience generated by the environment. In this section we describe only the core RL-Glue functions that are used to start an episode, perform one step of an episode, execute an episode, calculate the return from an episode and calculate the number of steps taken during an episode. We describe RL-Glue's extended functionalities later.

The `RL_start` function executes the first step of an episode. At the beginning of an episode the agent and environment must be set / returned to an initial configuration. The `RL_start` function performs the first step by calling `env_start` function, which returns an observation. Then `RL_start` calls the `agent_start` function, passing it the stored observation as input, and then returns control to the experiment module.

The `RL_step` function executes a single step of an episode (assuming the first step, `RL_start`, has already been taken). To take a step, RL-Glue passes the action, the agent (`RL_start` or `RL_step`) selected on the previous time step to the environment; this is achieved through a call to the `env_step` function with the action passed as a parameter. The new reward and observation produced by environment is then passed as input parame-

ters to agent_step. The new action returned by the agent is stored by RL-Glue until the next call to RL_step. In episodic tasks, the environment signals termination to RL-Glue by setting a Boolean flag in the observation. If termination occurs, RL-Glue calls the agent's agent_end function passing the final reward as input, ending the episode.

An experiment program can execute an episode by calling RL_start and then making repeated calls to RL_step until termination occurs. RL-Glue also defines several other functions used for more complicated learning experiments. The RL_episode function executes an episode. This function, however, assumes that the problem is episodic, i.e. the environment enters a special state called the terminal state, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. In a continuing task, experience must be cut off after some number of steps. RL-Glue provides another function designed for executing continuing tasks: RL_episode (steps) makes repeated calls to RL_step until steps time steps have elapsed or a terminal state is reached. These functions allow researchers to specify complex arrangements of training and testing phases.

RL-Glue stores the total discounted sum of rewards received during an episode and the total number of steps taken until termination is reached. The experiment can access the reward and steps data through calls to RL_return and RL_num_steps. These functions allow the experiment program to calculate average reward, cumulative reward, average number of steps and the minimum, maximum and variance of these values. Additional performance information and environmental data such as reward noise, graphics data and hidden state transition information can be collected in data files during execution or acquired through side calls to the environment.

6.2 RL-Glue Environments

The environment in the reinforcement learning framework completely describes the problem domain: the state transition dynamics, the reward function and the observation signal. In the RL-Glue protocol, an environment is described by the env_start and env_step functions. The env_start function selects the initial observation at the beginning of an episode. env_step contains the environment's state-transition dynamics: given an action,

`env_step` updates the environment's state and returns a new reward and observation. An environment need only implement these two functions to be compatable with the RL-Glue interface standard.

6.3 RL-Glue Agents

A reinforcement learning agent typically contains a learning algorithm and an action selection policy. In RL-Glue, an agent is defined by the `agent_start` and `agent_step` functions. `agent_start` selects the initial action at the beginning of an episode, based on an initial observation. `agent_step` selects a new action given a reward and observation. The `agent_end` function is called if the environment enters an absorbing terminal state, in episodic tasks. This function performs final learning updates based on the terminal reward but returns no action. In continuing tasks, `agent_step` is called until the `steps` counter in `RL_episode(steps)` elapses; `agent_end` is never called. An agent need only implement these three functions to be compatible with the RL-Glue interface standard.

6.4 RL-Glue Experiments

RL-Glue is a communication protocol: it provides a standard connection mechanism for learning agents and environments. A learning experiment, on the other hand, specifies the schedule of episode execution and the performance data to be collected. This high-level control cannot be incorporated into the agent or environment modules because the interface module has calling control. The agent and the environment cannot request RL-Glue to execute an episode or return the cumulative reward collected during the previous episode. Our approach employs a separate experiment module in the system architecture to control the execution of RL-Glue; this gives the experimenter complete control over the execution of RL-Glue and the experimental settings used for benchmarking agent programs on environments.

A typical experiment program might initialize RL-Glue with a call to `RL_init`, execute an experiment by making n calls to `RL_episode`, and finally call `RL_return` to get the cumulative reward achieved by the agent on the n^{th} episode. Experiment programs, however, are capable of specifying more complicated learning experiments. For example, a

experiment program could specify a multiphase schedule of training and testing over a finite set of experience trajectories through the environment's state space, while collecting the cumulative reward and the variance in the value function approximation. The experiment program is also used to process and display data summarizing the agent's performance on the task. The experiment program can gain access to performance data through calls to `RL_return` and `RL_num_steps` or data files written by the agent and environment at runtime. In general, the experiment program provides a simple way to control all aspects of a learning experiment.

6.5 RL-Glue Naming Conventions

The RL-Glue specification provides a set of standard function definitions: function names, input parameters and return values. The intended role of these functions is reflected by the definitions. The functions define the agent's, environment's and experiment's capabilities, but not how each should be implemented. The suite of RL-Glue functions facilitate a number of typical benchmarking tasks. This approach achieves the desired standardization with minimal restrictions on agent and environment programs.

The exact implementation details of each function are completely dependent on the needs of the experimenter. For example, a typical learning agent will update its value function in `agent_step`. This is not a requirement of `agent_step`, but merely an implementation of a specific agent. The `agent_step` function might instead perform a search of a game tree it had built during the previous episode. The `agent_step` function definition only requires that the agent take a reward and observation as input and return a new action on every step of the episode.

We will often describe the ambition of a function and provide examples of how it could be used, but these are only interpretations of particular instances of these functions. RL-Glue standardizes the names and definitions of these functions. Each implementation of an agent or environment function, however, provides the details of one specific instance. This principle applies to all the functions described in this work.

6.6 Practical Illustration

Up to now we have only described the core functionality of RL-Glue; the minimal components required to implement simple learning agents and execute experiments that describe performance in terms of cumulative reward and number of steps to goal. In this section, we will provide an implementation of the agent, environment and experiment modules to illustrate the simplicity and expressiveness of the RL-Glue specification. We provide full pseudo code for a tabular Sarsa(λ) agent [Sutton and Barto, 1998], a variation of the original Mountain Car domain [Moore, 1990] and an online cumulative reward benchmark.

The details of the Sarsa(λ) algorithm and the Mountain Car task are not of interest here. The objective of this example is to illustrate how a nontrivial learning agent using function approximation and eligibility traces can be applied to a classic control task using only the basic set of core RL-Glue functions.

6.6.1 Environment

In the Mountain Car domain, at the beginning of every episode, the car is repositioned at the bottom of the valley with 0 velocity. The following `env_start` function initializes the environment to the start state and returns the observation (which is equal to the environmental state in this domain).

---

name: env_start
input: {}
output: observation

position $\leftarrow -0.5$ #Environment global variables
velocity $\leftarrow 0$
return [position, velocity]

During every step of an episode, the environment updates the car's position and velocity based on the action selected by the agent. The actions "full throttle", "neural" and "full reverse" are mapped to 0, 1 and 2, respectively. The environment must then return the new observation, a reward, and a termination flag indicating if the agent has reached the goal position. Tuples are used to store the observation, reward and terminal flag.

name: env_step
input: action
output: observation

velocity $\leftarrow$ update_velocity(velocity, action)
position $\leftarrow$ update_position(position, velocity)
observation $\leftarrow$ [position, velocity]
if goal() **then**
  return $[-1$, observation, $true]$
else
  return $[-1$, observation, $false]$
end if

The goal function checks the position to determine if the car has reached the east side of the valley. These two functions fully specify the dynamics of the Mountain Car domain in RL-Glue.

6.6.2 Agent

The observation in the Mountain Car domain is real valued. To learn a state-action-value for the Mountain Car domain we must use function approximation to estimate the value of the continuous-valued observation. We use a simple function approximator to discretize the observations by aggregating values into discrete groups. We use Sarsa(λ) policy iteration to learn a value function over the aggregated observations and update our policy based on the estimated values. At the beginning of an episode, the agent must select an action based on the initial observation returned by the environment:

name: agent_start
input: observation
output: action

for all s, a **do**
  $e \leftarrow 0$
  $Q \leftarrow 0$
end for
$S \leftarrow$ box_val(observation) #S and A are global agent variables
$A \leftarrow$ select_action(S)
return a

The `box_val` function performs state aggregation and the `select_action` function selects actions according to an ϵ-greedy policy over state-action values [Sutton and Barto, 1998].

The following `agent_step` function implements a step of the Sarsa(λ) on-policy TD-control algorithm and returns a new action:

---

name: agent_step
input: reward, observation
output: action

$e[S, A] \leftarrow 1$
$s \leftarrow$ box_val(observation)
$a \leftarrow$ select_action(s)
$\delta \leftarrow$ reward $-Q[S, A] + \gamma Q[s, a]$
for all s', a' **do**
  $Q[s', a'] \leftarrow Q[s', a'] + \alpha\delta e[s', a']$
  if $a' = a$ **then**
    $e[s', a'] \leftarrow e[s', a']\alpha\lambda$
  else
    $e[s', a'] \leftarrow 0$
  end if
  $S \leftarrow s$
  $A \leftarrow a$
end for
return a

Mountain Car is an episodic task: when the car reaches the top of the east side of the valley the environment enters a terminal state. RL-Glue notifies the agent that the episode has ended by calling the `agent_end` function and passing the terminal reward as input:

---

name: agent_end
input: reward
output: $\{\}$

$e[S, A] \leftarrow 1$
$\delta \leftarrow$ reward $-Q[S, A]$
for all s', a' **do**
  $Q[s', a'] \leftarrow Q[s', a'] + \alpha\delta e[s', a']$
end for

6.6.3 Experiment

In order to benchmark an agent on an environment in RL-Glue, we must design the experiment program that specifies a learning experiment. For the Mountain Car domain we are interested in how quickly the agent learns and in the quality of the policy learned by the end of training. The reward in this task is -1 per time step; the agent should learn a policy that allows it to escape the valley in as few steps as possible. Thus, the cumulative reward over a fixed number of episodes will summarize how quickly the agent found a good region in the policy space and how good the learned policy is:

---

performance $\leftarrow 0$
$RL\_init()$
for $episode = 1 : 1000$ **do**
  $RL\_episode()$
  performance $\leftarrow$ performance $+RL\_return()$
end for
return performance$/1000.0$

This is a simple illustration of what an experiment program might look like and how a experiment program can make use of the functions specified by RL-Glue. Experiments can be much more complex in terms of episode schedule and data collection.

The above code illustrates how a classic control problem and a learning agent with function approximation can be implemented using only five of the core RL-Glue functions. Furthermore, the performance of our agent on the Mountain Car domain is measured using a seven-line experiment program. In the next chapter we will describe how the agent and environment programs can employ additional RL-Glue functions to encode more advanced learning agents and experiments.

6.7 Additional Functionality

The core set of RL-Glue functions may not be expressive enough to implement more complex agents, environments or evaluation schemes. The core set of RL-Glue functions do not provide a way for an agent program to adjust online to new domains. Another issue involves controlling the agent's trajectory through the state space of the environment. Ideally, the ex-

perience trajectory of the agent could be replicated independent of the learning algorithm or policy used. Finally, an experiment program should be able to conduct more traditional supervised learning experiments. The agent should be allowed some finite amount of training data to learn a policy with performance measured only during the testing phase on the test data.

In this chapter we describe the functions used to realize this additional layer of functionality. We provide intuition about how these functions could be used; the implementation details should be decided by the researcher.

6.7.1 Data Management

In reinforcement learning, we must repeat experiments many times to reduce variation in the results to provide evidence that the results generated by a learning algorithm are statistically significant. Ideally, each run is preformed under identical conditions. The `env_init` and `agent_init` functions are used to restore the environment to its initial state and reset the learning agents value function, model and/or policy to their initial conditions. The environment and agent init functions also have a more practical role of declaring and allocating any global variables or data structures used during the execution of a learning experiment.

Correspondingly, `env_cleanup` and `agent_cleanup` functions release memory resources allocated by the init functions.

6.7.2 Super Agents

Efficient generalization between tasks is an important topic in reinforcement learning. It would be very difficult to program a learning agent that can handle a variety of observation and action types without some assumptions about the dimensions or data types of these values. I have developed a task description language that provides the agent with information about the the environment before learning begins. The environment returns a Task_specification string to RL-Glue that describes observation, action and reward types and ranges. The Task_specification is then passed to the agent program before the learning experiment begins to allow the agent program to make adjustments for the current task or exit if the agent is not compatible with the environment. The current version of the task

description language encodes:

- the version of the task description language used
- if the task is episodic or continuing
- the number of action dimensions
- the types of each action dimension
- the ranges of each observation dimension
- the number of observation dimensions
- the types of each observation dimension
- the ranges of each action dimension

The environment's `env_init` function encodes information about the problem in a ASCII string. The Task_specification is passed from the environment, through the interface, to the agent as a parameter to the `agent_init` function. The agent may optionally ignore the Task_specification or use it to initialize its learning algorithm. A detailed discussion of the current version of the Task description language can be found in Appendix A.

6.7.3 Controlling Environment Dynamics

We often want to control the starting state and how stochastic state transitions are in the environment when generating results for publication or competition. This functionality would make replication of results easier. The randomness in the environment can be determined by thresholds used to make decisions on random events and the seeds passed to the random number generator(s). Setting the initial configuration of the environment limits the number of trajectories through the state space the agent can experience. This approach is more effective than random initialization because particular starting states may be more favorable or detrimental to solving a task. Starting near the goal in a navigation task or starting near an adversary in a game can greatly effect the performance of an agent. An experiment program can set the starting state and the randomness of the environment to ensure more uniform behavior during every episode.

Before the environment can be set to a particular state, the experiment program must

acquire a legal state from the environment. The full environmental state, however, could be too large to pass as a single parameter. Furthermore, we want to avoid giving the agent direct access to the environmental state. Instead, we can get a `state_key` from the environment. The `env_get_state` function returns a `state_key` so that the environment can be returned to the state later upon presentation of `state_key`. The `state_key` could in fact be the state object, or a hash value for the current state. We can then restore the environment to the state encoded in the `state_key` by calling `env_set_state(state_key)`.

Similarly, the randomness of the environment can be encoded in a key. By controlling the randomness of the environment, we are attempting to make all state transitions deterministic. The `random_seed_key` is accessed by calls to the environment's `env_get_random_seed` function. The `env_set_random_seed` restores the random seed used by the environment. The environment should produce the same sequence of states, given the same sequence of actions.

6.7.4 Standardizing Randomness

Different programming languages, operating systems and hardware architectures generate random numbers differently. Some generators produce long streams of uniform uncorrelated random numbers. Others generators are well known for having a short period sequences and can be disrupted by other programs executing at the same time.

In a competition setting, we want to guarantee that random numbers produced by the environment are the same regardless of the programming language or hardware used to run an experiment. Researchers using a particular programming language should not have an advantage in competition because they hacked the environment's random number generator. The `env_standardize_randomness` function is meant to ensure that agents benchmarked on an environment get the same randomness. For example, an environment program could store a long sequence of random values in a data file, which the `env_step` function queries each time step. This cross-platform standardization is key to establishing fair and uniform conditions for bake-off competitions.

6.7.5 Multiphase Learning and Evaluation

In supervised learning, the agent learns a classifier from a set of training data. The agent's performance is not measured during training, its classification error is only measured on the test set, while learning is disabled. In reinforcement learning, we may want to employ this multiphase evaluation methodology: the agent is given some finite amount of time to interact with the environment to learn a value function and policy. Then the environment is reset to some initial state and learning in the agent is disabled, temporarily freezing the agent's policy. Evaluating the "frozen" agent on several new trajectories through the state space will not only evaluate the quality of the agent's learned policy, but also the efficiency with which it explored the environment during the training phase.

RL-Glue implements multiphase evaluation in a simple way. An RL-Glue agent assumes that all experience from the beginning of time to the current time step is training data (from the initialization call to `agent_init`). The interface notifies the agent that training has completed and the testing phase has begun with a call to the `agent_freeze` function. This allows the agent to freeze its current policy, stop learning and/or end exploration. The call to freeze the agent comes from the experiment program. An experiment program can train the agent over several episodes (`RL_episode`), then call `agent_freeze`, reset the environment to some starting state (`RL_set_state`) and then execute a number of episodes while measuring the cumulative reward (`RL_return`).

Chapter 7

RL-Glue Software

The RL-Glue specification is language independent: the functions described in the previous chapter could be implemented in any programming language. The main goal of the RL-Glue communication protocol is to provide a mechanism to standardize the execution sequence that occurs during a learning experiment so that experimental results can be reproduced easily for comparison and competition. The current implementation of RL-Glue has been designed to support agents, environments and experiments written in any programming language. In this chapter we describe the software implementation of the RL-Glue protocol, illustrate how the architecture supports multi-language communication and discuss how the current implementation can be extended to support other languages.

Implementing a single-language version of RL-Glue involves developing the function-based architecture illustrated in Figure 6.1. This involves implementing the RL-Glue module containing the RL-Glue functions described in Section 6.1. An experiment program would contain a main method, call the interface functions and collect statistics based on calls to the interface (`RL_return` for example). Finally, the agent and environment modules would implement the mandatory interface functions: `agent_start` and `agent_step` for agents, and `env_start` and `env_step` for environments.

7.1 Multi-language Support

To extend the single-language implementation of RL-Glue to allow learning agents, environments and testing code to be written in any language, we must call functions and convert data types across languages. To achieve this kind of flexibility we implemented the multi-

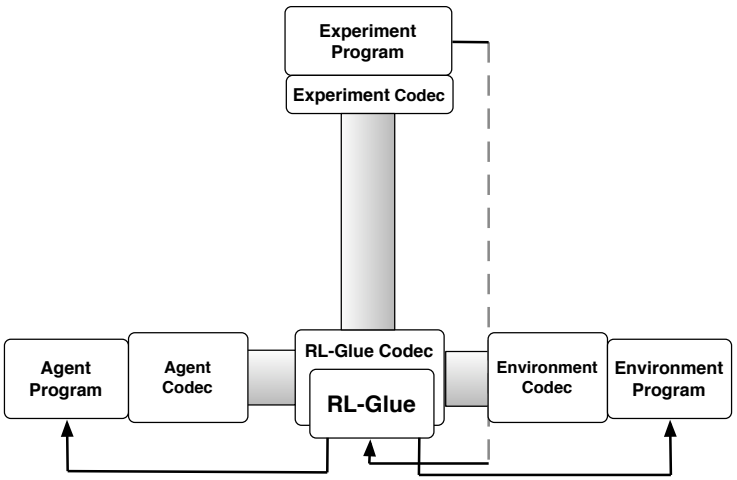

Figure 7.1: The RL-Glue pipe and network communication architecture. Arrows indicate direct language-to-language function calls.

language architecture depicted in Figure 7.1. The main difference between Figure 7.1 and the original interface specification in Figure 6.1 is the addition of codecs. A codec encodes and decodes data: it encodes data structures from a particular language to an ASCII string representation of the data and decodes ASCII strings to the appropriate data structures in a particular language. These codecs can be used to read and write data to pipe files or network sockets. In the discussion below, we describe how RL-Glue interacts with file pipes to achieve multi-language communication. Later describe how the system can be extended to communicate over a network.

We use file pipes with codecs to communicate function calls and data between different languages. A pipe file is a FIFO ASCII buffer. String data can be written to and read from the buffer, however, data read from the buffer is removed from the buffer. The RL-Glue codec, shown in Figure 7.1, is responsible for converting actions, observations, rewards, `state_keys` and `random_seed_keys` to ASCII strings from their C++ data types and *vice versa*. The RL-Glue codec makes function calls by writing the string names of agent and environment functions and the action, observation and reward strings to the pipes. The RL-Glue codec is language independent and need only be changed if additional functionality is added to the interface specification. The agent and environment codecs read the string function calls from the RL-Glue codec off the pipes. The agent and environment codecs must read actions, observations, rewards, `state_keys` and `random_seed_keys` off the

pipe and convert them to types corresponding to the language the agent and environments are written in, then make function calls to various agent and environment functions. The agent and environment codecs are not language independent. A codec must be written for any language that one wishes to write agent and environment code in. The experiment codec works in the same way as the agent and environment codecs.

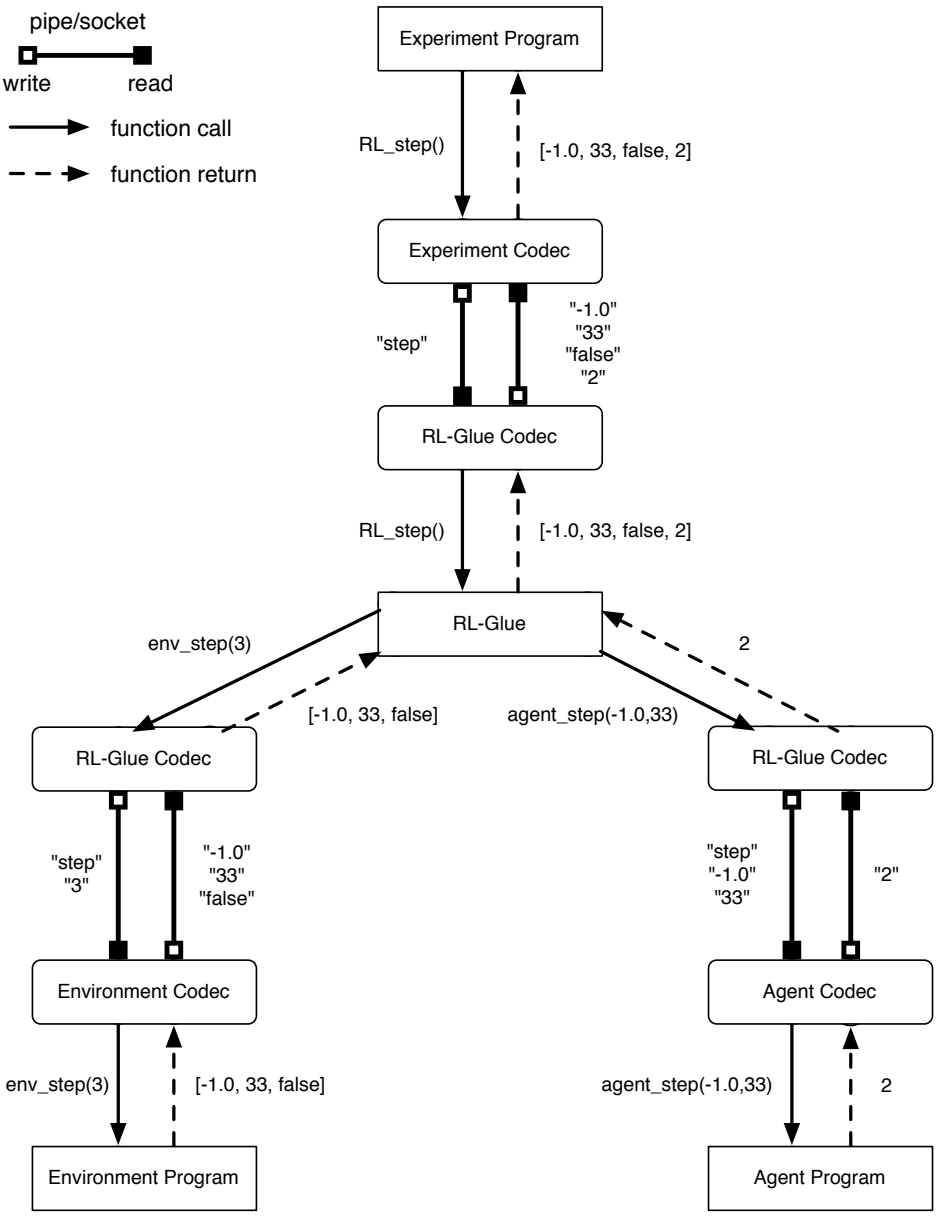

Figure 7.2: The executions sequence that occurs during a call to `RL_step` using pipe communication.

To illustrate our ASCII communication protocol we provide an execution trace of an

experiment using RL-Glue and this pipe communication scheme. Imagine the agent, environment and experiment programs in Figure 7.1 are all written in different languages and we wish to execute a single step of an episode. Figure 7.2 depicts the execution sequence that occurs during a call to `RL_step` from an experiment program. This example illustrates how a single step of an episode can be conducted by passing the string names of functions and data values encoded as strings across a pipe. In fact, all of the functions specified by RL-Glue are executed using the same ASCII interface. Whether the experiment program is executing a single learning step, collecting the average reward over a number of episodes or freezing the agent, the function names are converted to strings and data values are encoded and decoded to and from strings. This framework can support agents, environment and experiments written in any programming language.

RL-Glue is an open source project, meaning anyone can download and edit its source code. The current distribution of RL-Glue includes codecs for C++, Java, Python and Lisp. If a researcher wanted to write learning agents in Objective Caml, for example, they could simply implement a Caml codec for agents and be fully compatible with the existing code base. The software architecture was designed to encourage this kind of community driven growth. As new languages emerge and the reinforcement learning community's needs change, the code base of RL-Glue can be extended to new programming languages and paradigms.

7.2 File Pipes, Codecs and Network Communication

Network communication can be realized in RL-Glue using codecs and the same ASCII protocol used for pipe communication. If we imagine the links between modules (shaded bars) in Figure 7.1 as connections across a network, the description for executing a step of a learning experiment above is the same for calling `RL_step` across a network. The file pipe codecs are replaced with network codecs that read and write to sockets. The basic functionality of the RL-Glue software architecture, however, does not change: encoding program values in a string and decoding strings into values for a particular language.

The software architecture of RL-Glue enables 4 basic modes of execution: direct function call, pipe communication, network communication and mixed mode communication.

RL-Glue behaves exactly as described at the beginning of this Chapter (basic C++ implementation) when the agent, environment and experiment program are all implemented in C++: the experiment program makes direct calls to the functions in the RL-Glue module and RL-Glue directly calls functions implemented by the agent and environment programs. If the agent, environment and experiment programs are implemented in different languages then RL-Glue uses the pipe interface to facilitate multi-language communication. If the agent, environment and experiment programs are only accessible through remote servers, RL-Glue uses the socket interface to conduct learning experiments. The RL-Glue software architecture also allows a mixture of all three modes. The experiment and RL-Glue modules may use direct language calls, RL-Glue and the agent may use pipe communication and RL-Glue and the environment might communicate over a network connection. Any combination of the direct language call, pipe and network communication can be implemented with this architecture.

Network support allows agents to communicate with remote environments over the Internet, providing an ideal setting for competitions by separating the environments from agent writers. This separation helps with empirical validity, because agent programs do not run in the same memory space (or even the same machine) as the environments, eliminating many of the concerns regarding agents manipulating random number generators and performance data.

7.3 Design Decisions

It might seem natural to implement RL-Glue using an object-oriented approach to maximize the extendibility and flexibility of the software architecture. The object-oriented approach often results in elegant code that readily facilitates the future extension and growth of a software package; the latter is critical for the establishment and continued support of a software standardization system like RL-Glue. This extendibility does not come for free. Agent and environment programs would have to use special object-oriented language features, which are not available in all languages. This language constraint opposes one of the main design goals of RL-Glue: the system should be light-weight and compatible across as many platforms as possible. Future versions of RL-Glue may move toward a more object-oriented

approach, however, our design goals are well served by a simpler function-based system.

Chapter 8

RL-Library

The UCI Machine Learning repository not only standardizes the interface between algorithms and domains but also provided a centralized library where researchers from around the world can download existing data-sets used in publications and upload new domains based on recent publications. RL-Glue provides a platform on which to benchmark agents and environments regardless of the implementation language, operating system, hardware or physical location. The empirical standardization provided by RL-Glue must be coupled with a publicly accessible library of code to achieve the goal of exact reproducibility of results for publication and competition. The University of Alberta Reinforcement Learning Library, like the UCI, SPEC, NASA, and Matrix-Market databases, was created to provide a library of benchmarking resources, compatible with RL-Glue, for the reinforcement learning community.

Researchers can download the code used to generate results found in publications, and also submit their agents, environments and experiment programs used in their work. RL-Library is meant to be the primary repository of RL-Glue compatible software for the machine learning community. In the remainder of this chapter, we describe the layout of RL-Library, its core functionality and how RL-Glue and RL-Library can be used for benchmarking in reinforcement learning.

8.1 Library Structure

The Library contents are divided into four main sections or shelves: the agent shelf, the environment shelf, the experiment shelf and the project shelf. Each entry on each shelf

consists of a text description, a list of contents, a version number and a download link. The agent download contains all the code used to implement a particular learning algorithm. For example, a tabular Sarsa agent might consist of the source code and a library of helper functions used by the agent. Similarly, each environment distribution contains all the necessary code used to implement the problem. Each experiment includes the experiment program and any statistic collection and graphing code. The Sarsa agent, Mountain Car task, and experiment programs presented in Section 6.6 are stored in the library as single file downloads on the agent, environment and experiment shelves respectively.

The project shelf contains multi-file distributions that will typically contain programs that are also stored on other shelves of the library. Each project in the RL-Library contains all the necessary files used to generate results for a publication. A project is meant to be a complete package that allows one to exactly reproduce the graphs and tables used in a particular publication. A project typically includes the agent, environment, benchmark, statistics code, graphing code, parameter settings, makefiles, scripts and anything else used to generate experimental results. Project distributions set the performance standard for a particular environment and benchmark. The project shelf lets other researchers confirm benchmark results and benchmark other algorithms against the ones used in the project under identical experimental conditions.

RL-Library provides individual distributions for all agents, environments, experiments and projects in the library. Furthermore, the entire contents of each shelf are available so that users can easily acquire all the agents or environments in the library, for example. This layout provides efficient public access to learning agents and environments developed by researchers around the world, for research and teaching.

8.2 RL-Libary: the living entity

The RL-Library shelves are stocked by the reinforcement learning community–any researcher can act as a librarian. Ideally, researchers will add the agents, environments and experiments, in an *ad hoc* fashion, when they make new advances and then later upload project distributions used to generate results for conference and journal papers. The test environments used and agents developed for annual benchmarking competitions and machine

learning classes will help maintain a steady stream of new programs to ensure the continued growth and stability of RL-Library.

The library can be used to present challenge problems to the reinforcement learning community. A researcher can upload a new environment that highlights one or more of the known limitations of current methods and challenge the community to tackle these issues in a competitive fashion. This approach helps focus the research efforts of a large heterogeneous community on open problems, accelerating research progress.

The library allows researchers to upload source code using simple online forms. The code is then subjected to a brief review process to ensure the code is technically sound and to determine the submission's contribution. The library provides an efficient, public way to disseminate standard versions of agents and environments for publications, competitions and instruction for the reinforcement learning community.

Chapter 9

Standardizing Mountain Car

RL-Glue and RL-Library give us the power to begin addressing the standardization issues exhibited by the Mountain Car domain. In this chapter we formulate and implement a standard version of the Mountain Car problem under the RL-Glue specification. We then benchmark several algorithms on the Standard Mountain Car domain and present, for the first time in reinforcement learning, a set of benchmarks which are publicly available through RL-Library and completely reproducible. This chapter sets a first benchmark for the Mountain Car domain and provides a comprehensive illustration of how RL-Glue and RL-Library can be used to standardize empirical analysis. All the code, used to generate the results presented in this chapter, can be found on the RL-Library web-page: *http://www.rlai.cs.ualberta.ca/RLR/*

9.1 Problem Specification

In this section we present a standard formulation of the Mountain Car test domain using RL-Glue. We use a variation of Moore's [1990] original problem. The car's movement is is governed by the following equations:

$$position \quad += \quad velocity; \;\; -1.2 < position < 0.6$$

$$velocity \quad += \quad 0.001 \cdot action + (-0.0025 \cdot cos(3 \cdot position)); \;\; -0.7 < velocity < 0.7$$

The system is fully characterized by the continuous observation variables position and velocity. The action input is discretized into 3 values: full reverse (0), neutral (1) and full throttle (2). A reward of -1 is given every step and termination is reached if and only if

the car passes the top of the east side of the valley with a nonzero velocity ($position > 0.6$ and $velocity > 0$). The car's position and velocity is reset to the boundary values if the car achieves a position or velocity outside their respective ranges. At the beginning of an episode, the car is initialized to a legal random position and velocity.

In the following, we refer to this random starting position variation of the Standard Mountain Car domain as the MC_Random environment.

9.2 Solution Methods

We use three different learning agents on MC_Random: Tile_Sarsa(λ), Tile_Q(λ) and Tile_AC(λ). We used the Sarsa(λ) and Watkins Q-learning algorithms with tile-coding function approximation as described in the Reinforcement Learning text [Sutton and Barto, 1998]. The Tile_AC(λ) agent is based on the tabular Actor-critic algorithm in the same text. The extension of the Actor-Critic method to the linear function approximation case involves implementing TD value iteration with replacing eligibility traces [Singh and Sutton, 1996] for the Critic and integrating Gibbs action selection over approximate state-action values for the Actor [Sutton and Barto, 1998]. We used linear tile coding as our function approximation method.

The learning rate, α, was set to 0.5 and the temporal credit assignment parameter, λ, was set to 0.95 for both Tile_Sarsa(λ) and Tile_Q(λ). Singh and Sutton found these parameter settings performed the best in their study of a similar variant of the Mountain Car domain [Singh and Sutton, 1996]. We performed a similar study to determine a good set of parameters for the Tile_AC(λ) agent. Figure 9.1 shows the performance, number of steps to goal, of Tile_AC(λ) on MC_Random for various combinations of α, β and λ. From these graphs we can see that $\lambda = 0.9$, $\alpha = 0.51$ and $\beta = 0.2$ achieved the lowest average number of steps to goal. The exploration parameter, ϵ, was set to zero for all three methods. Instead, we initialized the state-value, state-action-value and action preference functions optimistically to zero to encourage exploration. Finally, we used ten 9 x 9 tilings for tile coding for all three agents. We tiled the position and velocity together for each action, with tile widths of 0.2125 and 0.0175 respectively.

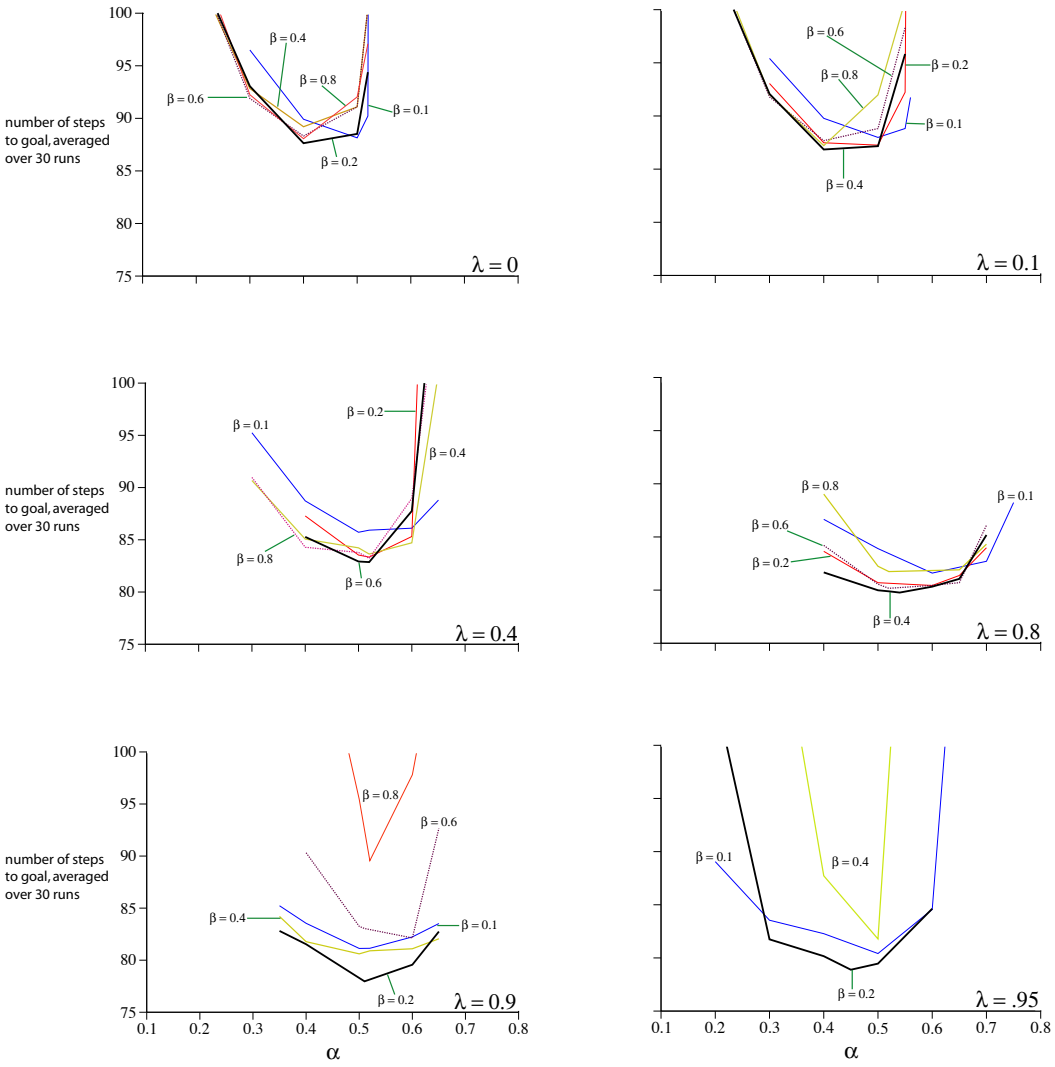

Figure 9.1: Average number of steps to goal for Tile_AC(λ), on MC_Random, for various combinations of α, β and λ. The bold line, in each plot, indicates β value that achieved the best performance for each value of λ.

9.3 Experimental Design

All the results presented in this chapter were generated by two experiment programs: Figure 9.1 and Table 9.1 generated by one experiment program and Figure 8.2 by the other. Each of the three agents were tested over 200 consecutive episodes. No upper limit was imposed on the number of steps taken during each episode. The results were then averaged over 100 independent runs.

9.4 Results

To establish a benchmark, one must provide adequate performance measures that best characterize the performance of a learning algorithm on a particular environment. We must establish that a given algorithm is superior in a variety of situations. In the Mountain Car task, the speed of learning and the quality of the final solution are the most important metric. Both of these, can be characterized by the number of steps to goal per episode.

	Average number of steps to goal	Time per batch (micro seconds)
Tile_Sarsa(λ)	91.5441	0.320119
Tile_Q(λ)	86.7475	0.33552
Tile_AC(λ)	79.2767	0.520472

Table 9.1: Long-run benchmarks for Tile_Sarsa(λ), Tile_Q(λ) and Tile_AC(λ) on MC_Random.

Table 9.1 illustrates the relative performance of all three methods. We can see that the Tile_AC (λ) achieves the lowest average number of steps to goal, but also takes the longest in terms of CPU time. The time difference between the methods, however, is less than one second for a batch of 200 episodes and is thus negligible in this case. The Tile_AC(λ) appears to be superior.

The averages in Table 9.1 do not always provide the best indication of performance. It is difficult to assess learning speed, performance variance and determine if an algorithm achieves steady-state performance on a task using long-run averages. Figure 9.2 presents the learning curves (average number of steps to goal) for each of the three methods on MC_Random. We binned the data into 10 episode bins to smooth the learning curves. In

these plots we can see that Tile_AC(λ) learns faster than Tile_Sarsa(λ) and Tile_Q(λ) and achieves much lower variance, although all three methods converge to approximately the same quality policy (after training all three control policies require the same number of steps to goal, on average, from random starting positions).

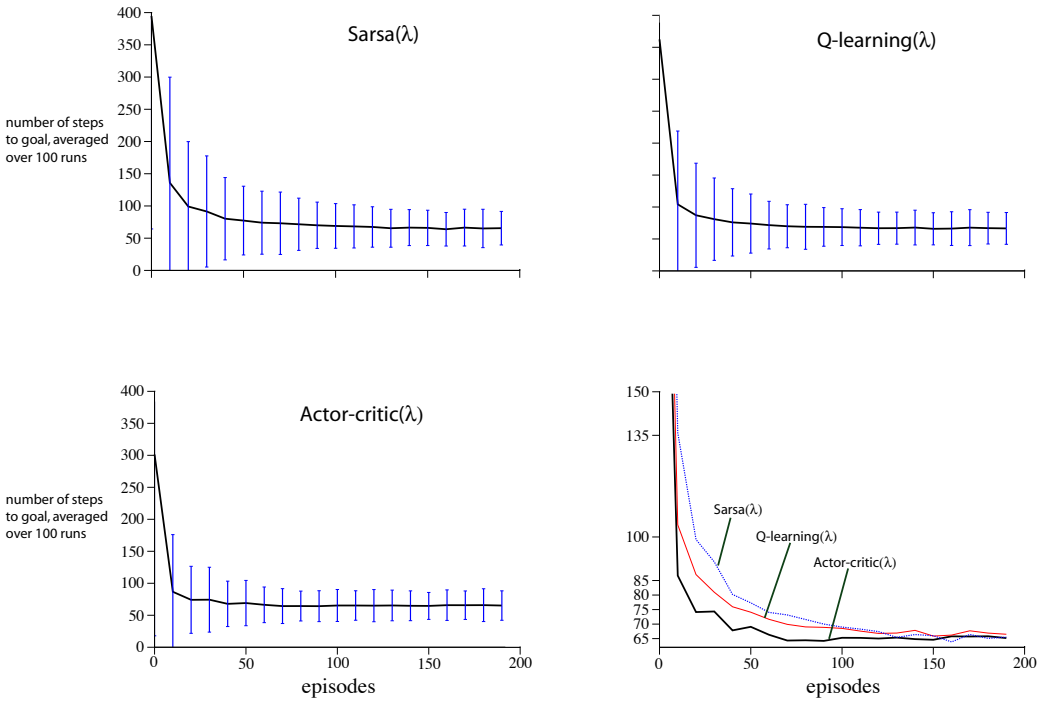

Figure 9.2: Learning-curve benchmarks for Tile_Sarsa(λ), Tile_Q(λ) and Tile_AC(λ) on MountainCar_RS. Results averaged over a 10-episode bin.

Figure 9.2 and Table 9.1 are the first benchmarks for the Mountain Car control task. The number of steps to goal achieved by Tile_AC(λ) on MC_Random (average number of steps to goal Table 9.1) establishes the state-of-the-art for this variation of the Mountain Car domain. This benchmark will stand until another researcher can illustrate better performance, as in Table 9.1, using the same environment and experiment programs from RL-Library.

9.5 Discussion

RL-Glue provides immediate feedback on an algorithm's performance on a given task, through a variety of performance measures based on thousands of simulated agent, environment interactions. Figure 9.2 and Table 9.1 illustrate that each learning agent converges

to some suitable representation of the value function, yielding a near optimal control policy. All necessary performance data was accessed through interface calls (`RL_return`) and system calls (C++ timing routines). Each experiment (200 episodes and 100 independent batches) took approximately 1 minute, wall-clock time, to complete. This execution efficiency and empirical clarity makes RL-Glue a good platform for benchmarking and development.

The RL-Glue and RL-Library reduces the confusion regarding problem specification, experimental conditions and software versions. Any researcher can compare his or her results with existing results in the literature by downloading the necessary code from RL-Library. Researchers can easily compare results against existing benchmarks or establish new benchmarks and upload their code to RL-Library, removing the confusion surrounding choice and comparison of different performance metrics. By establishing a publicly available library of code, we have eliminated many of the standardization and evaluation problems in the literature (highlighted in Table 5.1).

Chapter 10

A Benchmark Suite for Reinforcement Learning

In the previous chapter we formulated a standard version of the Mountain Car domain and presented the first publicly available and reproducible results for several reinforcement learning problems. The Mountain Car benchmark was used to provide a proof of concept for RL-Glue and RL-library. In this chapter, we present benchmarks for ten test domains, using the same evaluation methodology established in the previous chapter. In this chapter, however, we provide only high-level descriptions of each environment and the agents that performed best on each task. The implementation details of each agent and environment can be found in RL-Library. The ambition of this chapter is to establish benchmarks and introduce standardized implementations of several classic reinforcement learning control tasks.

Our benchmarks are based on ten environment programs: three grid-world problems, a card game, a discrete sensor network, a random MDP, the Acrobot and three variations of the Mountain Car domain. These problems were selected to cover a large class of domains on which reinforcement learning methods are typically applied. These ten environments feature tabular state and actions spaces, continuous observations, multi-component actions, partially observable states, high dimensional action spaces, stochastic dynamics, non-stationary dynamics and continuing tasks. These environments highlight many open research areas in reinforcement and machine learning.

The following sections present benchmarks for each of the ten environments. For brevity, we do not present graphs for each algorithms learning curve, performance under dif-

ferent parameter settings, nor the results of other algorithms. Instead, we present summary statistics for the agent that performed best on each environment. Each agent-environment pair was benchmarked over n consecutive episodes using the same experiment program, where n corresponds to the number of episodes listed in each benchmark table. The results for each benchmark were then averaged over 100 independent runs.

10.1 Grid-world Benchmark

In the Grid_Mines environment, each two-dimensional map is randomly generated with several solid obstacles, several mines and a start and goal position. A reward of -1 is assigned on each time step, unless the agent hits a mine, resulting in a large negative reward. The agent's objective is to navigate from the starting state to the goal as fast as possible, without hitting mines. We used a tabular off-policy TD control agent with planning, DynaQ, on the Grid_Mines task. This agent, based on the DynaQ algorithm from the Reinforcement Learning text, builds a model of the environments state transition dynamics [Sutton and Barto, 1998]. Table 10.1 summarizes DynaQ's performance on the Grid_Mines.

Environment	Agent	Episodes	Average Number Steps
Grid_Mines(20)	DynaQ	1000	38.108910

Table 10.1: Grid-world benchmarks. The Grid_Mines(20) environment uses a maze of size 20×20.

10.2 General Cat and Mouse Benchmark

In the CM_Random environment, there are several solid obstacles, several stationary pieces of cheese, one mouse hole and a non-stationary cat. The cat moves (after the mouse moves) to minimize its distance to the mouse, choosing randomly between equal quality moves. The cat cannot see or move to the mouse if it is hiding in its hole. Each map is randomly generated at the beginning of each trial, but does not change between consecutive episodes. The agent gets a small positive reward for each step that the mouse occupies the same grid

space as a piece of cheese and a large negative reward if the cat and mouse occupy the same grid space. An episode ends when the cat catches the mouse. The agent's objective is to navigate the mouse through the maze, collecting as much cheese as possible while avoiding the cat. We used a tabular on-policy TD control agent, tabular_Sarsa(λ) based on the Sarsa(λ) algorithm described in Reinforcement Learning [Sutton and Barto, 1998]. The tabular_Sarsa(λ) agent maintains an estimate of the state-action-value function for each state-action pair and selects actions according to an $\epsilon-$greedy policy. The episodes were cut off after 1000 steps to avoid infinite episodes: an agent that learned to avoid the cat (with exploration disabled) could move the mouse about the maze forever, never getting caught. No negative reward was associated with this cut-off point. Table 10.2 summarizes the performance of tabular_Sarsa(λ) on CM_Random.

Environment	Agent	Episodes	Cumulative Reward	Average Num Steps
CM_Random(10)	tabular_Sarsa(λ)	5000	$4.68 \cdot 10^8$	846.237

Table 10.2: The General Cat and Mouse Benchmark. The CM_Random(10) environment uses a maze of size 10×10.

10.3 Schapire's Cat and Mouse Benchmark

The CM_Schapire environment is based on Robert Schapire's version of the cat and mouse problem. The dynamics are the same as CM_Random environment, except CM_Schapire uses the static grid-world depicted in Figure 10.1; every episode uses the same map configuration. We used the DynaQ agent. Episodes were cut-off after 1000 steps to avoid infinite episodes. The performance of DynaQ on CM_Schapire is summarized in Table 10.3.

Environment	Agent	Episodes	Cumulative Reward	Average Number Steps
CM_ Schapire	DynaQ	1000	23787038.16	967.225600

Table 10.3: The Schapire Cat and Mouse Benchmark.

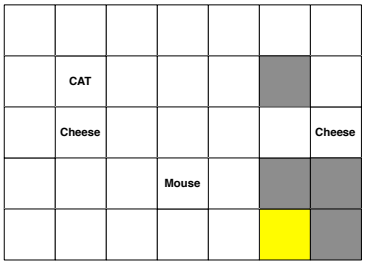

Figure 10.1: Schapire's Cat and Mouse world: The cat and mouse can occupy any of the 31 blank squares (dark gray squares indicate obstacles), except for square (4,5) which can only be occupied by the mouse. The cheese never moves.

10.4 Blackjack Benchmark

The Blackjack environment encodes the two-player game where the agent plays against the dealer. The agent must choose to "hit" or "stick" based on its cards and the dealers face-up card to achieve a sum of cards as close to 21 as possible. The dealer will "hit" until its card sum is 17 or greater. The agent is allowed to hit on 21 to avoid episode termination in the starting state. The agent gets a reward of $+1$ for a win, 0 for a tie and -1 for a loss. The agent's objective is to maximize its number of wins. This variation of the game is based on the Blackjack environment described in the Reinforcement Learning text [Sutton and Barto, 1998]. Table 10.4 presents the benchmarks for the tabular_Sarsa(λ) agent on the Blackjack environment.

Environment	Agent	Episodes	Average Reward	Average Num Steps
BlackJack	tabular_Sarsa(0)	100000	-0.193490	1.037769

Table 10.4: The Blackjack Benchmark.

On initial inspection, the Blackjack benchmarks seem low. Since Blackjack gives a reward of -1 for a loss and $+1$ for a win, an average reward of -0.193490 indicates the agent is losing more than 60% of the time. In fact, the tabular_Sarsa(0) agent achieved a winning percentage of only 38%. This is, however, much better than it sounds because the optimal Blackjack strategy described by Thorp illustrates that, given optimal play, a player cannot win more than 49% of games.

10.5 Sensor Network Benchmark

In the discrete sensor network problem, Sensor_Net, there are two parallel chains of sensors (see Figure 10.2). The area between the sensors is divided into cells. Each cell is surrounded by four sensors that can aim at a target that lies within the cell. Each sensor can perform three actions: aim at the target in the left cell, aim at the target in the right cell or the take null action. If three sensors simultaneously aim at one target the target's energy level is decreased; targeting is completely deterministic. A target is removed from the network when its power level reaches zero. The agents must control all N sensors and destroy all m targets as fast as possible. Although this problem has a relatively small state space $|s| = \sum_{i=0}^{m} \left[\binom{\frac{N}{2}-1}{i} 3^i \right]$, it has a high dimensional action space (N) resulting in N^3 possible actions. The Sensor_Net environment was implemented by Nikos Vlassis's research group at the University of Amsterdam. We used a factored tabular off-policy TD control agent, Factored_Q(0), on the Sensor_Net task. The Factored_Q(0) agent, also implemented by Vlassis, uses Q-learning to learn a state-action-value function for each of the N action dimensions. The performance of Factored_Q(0) on Sensor_Net is summarized in Table 10.5.

Environment	Agent	Episodes	Average Reward	Average Number Steps
Sensor_Net(8,2)	Factored_Q(0)	2000	3.084766	7.602015

Table 10.5: The Sensor Network Benchmark. The Sensor_Net(8,2) environment features 8 sensors and 2 targets.

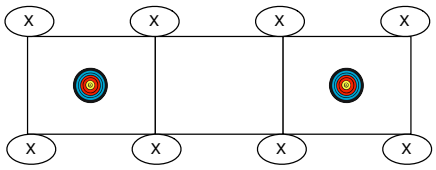

Figure 10.2: Vlassis's Discrete Sensor Network: A sensor network configuration with eight sensors (X) and two targets. Adapted from [Littman et al., 2005].

10.6 GARNET Benchmark

The GARNET environment is a randomly generated MDP (random states, actions and rewards) with probabilistic state transitions and a non stationary element: every k iterations the transitions are changed by randomly deleting n state connections and creating n new links between previously unconnected states. The agent is provided with a random binary observation vector that is not large enough to uniquely identify states. The agent's objective in this continuing task is to locate and try to stay in regions of the MDP that result in the highest average reward. We used an on-policy TD control agent with linear function approximation, linear_Sarsa(0) on the GARNET environment. The linear_Sarsa(0) agent is similar to Tile_Sarsa(λ), described in Chapter 9, except it does not use eligibility traces (TD(0)) and uses binary feature vectors generated by GARNET environment itself, instead of tile indices produced by tile coding. The linear_Sarsa(0) agent was allowed 100000 time steps to interact with the GARNET environment. Table 10.6 reports the average reward of the linear_Sarsa(0) agent on the GARNET environment.

Environment	Agent	Number of Episodes	Average Reward
GARNET	linear_Sarsa(0)	1×10^{11}	1.473955325

Table 10.6: The GARNET Benchmark.

10.7 Acrobot Benchmark

The Acrobot environment describes a two-link, under-actuated robot, roughly analogous to a gymnast swinging on a high bar (see Figure 10.3). The first joint cannot exert torque, but the second joint can. The agent must select actions of "backward", "null" or "forward" based on a four-dimensional continuous observation. The objective of the acrobot domain is for the agent to swing the tip (the "feet") above the first joint in the shortest amount of time. The Acrobot environment is based on the description provided in the Reinforcement Learning text [Sutton and Barto, 1998]. Table 10.7 presents the benchmark for the Tile_AC(λ) agent, used in the Mountain Car Benchmark (Chapter 9), on the Acrobot envi-

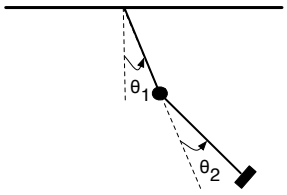

Figure 10.3: The Acrobot. Adapted from [Sutton and Barto, 1998].

ronment.

Environment	Agent	Episodes	Average Number Steps
Acrobot	Tile_AC(λ))	2000	61.168580

Table 10.7: The Acrobot Benchmark.

10.8 Delayed Mountain Car Benchmark

The MC_Delay variant of the Mountain Car problem is based on MC_Random, described in the Chapter 9. The MC_Delay environment is a non-Markov problem: the observations produced by the environment on time step t are actually the observation of the system that occurred k steps ago. The first k observations at the beginning of the episode are randomly generated and the agent does not receive the last k observations at the end of the episode. The agent must adjust its value estimates based on the delay, while making blind decisions on each time step. The actions, rewards and state-transition dynamics of the MC_Delay environment are identical to the MC_Random environment. We used an off-policy TD control agent with function approximation and static options, O_Tile_Q(λ), on the MC_Delay environment. The O_Tile_Q(λ) agent is based to the Tile_Q(λ) agent described in Chapter 9. O_Tile_Q(λ), however, employs static options corresponding to full forward for 5, 10, 20 and 50 time steps and full reverse for 5, 10, 20 and 50 time steps. These macro actions correspond to options whose initiation set contains every legal state. Termination of any particular option occurs, with probability 1, when the step timer elapses [Sutton et al., 1999]. The results for the O_Tile_Q(λ) agent on the MC_Delay environment are summarized in Table 10.8.

Environment	Agent	Episodes	Average Num Steps
MC_Delay(20)	O_Tile_Q(λ)	10000	119.389704

Table 10.8: The Delayed Mountain Car Benchmark. The MC_Delay(20) environment features observations that are delayed by 20 steps.

10.9 Stochastic Mountain Car Benchmark

In the MC_Stochastic variant of the Mountain Car problem (based on MC_Random), the agent's actions are not deterministic: the velocity generated by a positive force will be positive, but is perturbed randomly by positive noise and similarly for a negative force. In this formulation, the agent must account for these stochastic effects and learn a policy that is robust to small variations in action outcomes. We used the O_Tile_Q(λ) agent on the MC_Stochastic environment. Table 10.9 summarizes the results.

Environment	Agent	Episodes	Average Num Steps
MC_Stochastic	O_Tile_Q(λ)	5000	228.931504

Table 10.9: The Stochastic Mountain Car Benchmark.

10.10 Non-stationary Mountain Car Benchmark

The MC_Nonstat variant of the Mountain Car problem is a non-stationary domain where the force of gravity is adjusted every k steps by some random amount (positive or negative). Furthermore, to reach the goal, the agent must stop the car at the top of the hill with near zero velocity. If the agent drives through the goal region, a large negative reward is assigned and the episode terminates. The agent must learn to track the gravity change so that it does not drive off the end top of the mountain. We used the O_Tile_Q(λ) agent on the MC_Nonstat environment. Table 10.10 summarizes the results.

Environment	Agent	Episodes	Average Num Steps
MC_Nonstat(50)	O_Tile_Q(λ)	5000	142.003272

Table 10.10: The Non-stationary Mountain Car Benchmark. The The MC_Nonstat(50) environment changes the force of gravity every 50 episodes.

Chapter 11

Conclusions and Future Work

In this thesis we presented a communication protocol for reinforcement learning agents and environments. This protocol is designed to be the standard protocol for benchmarking agent and environment programs for publications and competitions in reinforcement learning. Our protocol 1) guarantees exact reproducibility of the execution sequence of a learning experiment, 2) enables plug and play interchanging of environments and agents, 3) is general and powerful yet non-intrusive, 4) is easy to convert existing agents and environments to.

RL-Glue's function-based interface guarantees the same sequence of function calls occurs during every call to `RL_episode`; results can always be replicated. RL-Glue's function-based architecture allows researchers to write generalized learning agents that are applicable to a wide variety of tasks. Every agent and environment implements the same set of basic interface functions. Furthermore, the task description language, introduced in this work, provides agent programs with information regarding the state and observation space, before a learning experiment begins to facilitate the creation of more general learning agents. The core set of agent and environment functions can encode a variety of common learning experiments, such as an on-policy TD control method, the Mountain Car domain and an online average reward benchmark. RL-Glue also features several additional agent and environment functions that control the environment's trajectories through the state space, standardize randomness for competition and allow the agent to be evaluated using complex sequences of training and testing phases. Agent and environment programs need only implement the core set of basic interface functions. The remainder of the agent

and environment code is completely free form. Researchers can quickly make their agent and environment programs compatible with the RL-Glue standard without changing their original code structure or style.

We also presented a software architecture for RL-Glue that facilitates network communication and is extendable with agents, environments and experiments written in any programming language. We designed the software implementation to 1) be light-weight with layered functionality, 2) support multiple programming languages and 3) support agent and environment interaction across a network.

Like the protocol on which its based, the software implementation of RL-Glue requires very little "mandatory" code. Agent and environment programs must employ the standard function names and data types, but are completely unrestricted otherwise. The interface code itself is efficient and does not require significant computational resources to produce benchmark results. RL-Glue facilitates multi-language communication using a simple ASCII string protocol and several language specific codec modules. This system allows the interface to mature and develop with the needs of the reinforcement learning community. Finally, the current implementation of RL-Glue supports communication across a network connection using the same ASCII protocol used for multi-language support. This allows agents to be benchmarked on environment programs that are only available through remote servers. Network communication makes RL-Glue a good platform for competitions because it isolates the learning agent from the environment and benchmarking code.

This thesis also introduced the University of Alberta Reinforcement Learning Library, a library of agent, environment and experiment code compatible with RL-Glue. RL-Library provides a public means to distribute standardized implementations of problem domains and state-of-the-art learning algorithms to the reinforcement learning community. Furthermore, RL-Library allows agent, environment, experiment and project code to be added to the library for review and shelving. Researchers will now be able to recreate results from the literature and test new learning algorithms using the same experimental settings used in publications. Instructors can provide their students with sample agents and environments for teaching and evaluation purposes without having to implement these methods themselves. The RL-Library was designed to be the primary repository for reinforcement

learning agents and environments, like the UCI database for supervised learning.

We illustrated the advantages of RL-Glue and RL-Library with a detailed case study on the classic reinforcement learning control task, Mountain Car. We surveyed five implementations of the "Standard" Mountain Car problem. Each differed substantially in either initial state of the environment, reward function or dynamics. We found similar standardization problems in the literature. We presented the first benchmark for the Mountain Car control task. This benchmark provides a performance baseline for all other learning agents to be measured against and a standardized implementation of the Mountain Car problem. RL-Glue and RL-Library directly address the standardization and evaluation questions raised in the Mountain Car survey: Which implementation should be considered the standard? How can one decide what version to use when testing new algorithms? Should each researcher implement his or her own code based on experimental descriptions in the literature? Which performance metrics should be used to evaluate advances in new publications?

We also presented the first suite of benchmarks for a number of classic reinforcement learning control tasks. For the first time, the reinforcement learning community has access to a set of standardized implementations of several control tasks and a corresponding set of performance baselines. All the environment, agent and experiment programs are publicly available in RL-Library for other researchers to use to verify, compare against and improve these benchmark results. Tables 10.1 to 10.10 are the realization of the goals and objects for RL-Glue laid out in Chapter 3: 1) facilitate the creation of benchmarks for reinforcement learning, 2) establish a suite of standard versions of benchmark problems, 3) set performance baselines for algorithm development, 4) facilitate accurate comparisons between algorithms on standard benchmarks, 5) allow researchers to replicate results from the literature, and 6) remove the need to re-implement others' code.

RL-Glue and RL-Library have been instrumental in establishing annual competitions where researchers from across the world develop agents for a variety of challenge problems. RL-Glue was used in the first bake-off (noncompetitive event) at the International Neural Information Processing Systems Conference in 2005 and the first reinforcement learning competition at the International Conference on Machine Learning in 2006. These competitions help focus research on one of the long standing goals of reinforcement learning

research: improving the reliability and power of learning techniques to the level required for real world problems. The work described in this thesis establishes a standard communication system for competitions and a library of compatible code, future competition organizers and participants can improve the practice of applying reinforcement learning methods. Ultimately, this approach may result in more applications to industrial problems such as, autonomous-vehicle control, hydraulic-dam control and hybrid-car fuel optimization providing concrete evidence that reinforcement learning algorithms are viable alternatives for real world tasks.

Much of the future work on RL-Glue and RL-Library will be a community effort. The current software implementation of RL-Glue must be extended to support the full spectrum of programming languages used by the reinforcement learning community. The code base of RL-Glue will grow over time and adapt to the community's needs. It is, in fact, necessary for the continued growth and development of RL-Glue that we allow the community to extend or re-implement the software architecture. The RL-Glue interface is a standard communication protocol that facilitates the establishment of benchmarks for reinforcement learning. The current software implementation, however, is just that. It is important that RL-Glue continues to provide reproducible benchmark results and multi-language support: the implementation details can and will change. A key factor in the growth of RL-Library will be the rate at which conference and journal reviewers begin to accept RL-Glue as the standard for reinforcement learning and begin requiring new publications to illustrate performance on the standard benchmarks. RL-Library has already been embraced by several members of that community: a number of agents and environments featured in the reinforcement learning competitions have been submitted to the library over the past few months. Papers appearing in international conferences have also used agent and environment programs from RL-Library. RL-Library will grow as more competitions are held, more classes make use of it and more researchers begin to publish results on the reinforcement learning benchmarks.

There are a number of other things to be done to further establish RL-Glue and RL-Library as the standard benchmarking tools in reinforcement learning. One involves establishing a large suite of benchmark challenge problems compatible with the interface

standard. These problems must be formulated so that they highlight the current algorithmic challenges in reinforcement learning and also reflect many of the open problems in AI. For instance, consider a hybrid gas-electric car. A control mechanism must choose between using the gas engine or the electric motor to supply the power requested as a driver pushes the accelerator. An agent must choose how much of the requested power will come from the motor and how much to drain the battery, based on various sensations such as car velocity, acceleration, engine temperature, outside temperature and battery level. This problem is interesting because of it's continuous state and action space, and the delay between decision-making and the measurable effects of actions. The public availability of these challenge domains, in RL-Library, will foster interest in reinforcement learning within the large AI community and lead to advances in some of the fundamental scalability issues in machine learning.

The next step will focus on algorithm development to address a number of the open problems in reinforcement learning, such as dealing with high dimensional state and action spaces, making continuous valued decisions, speeding up learning and dealing with delayed feedback and irrelevant information. Significant advances can be made in these areas through careful analysis of problem input. If we can account for the correlations among each of the agent's sensations and between sensations and the received reward signals, we may be able to eliminate irrelevant information, while placing more weight on unlikely events with significant consequences on system behavior, thus increasing learning efficiency. Novel combinations of intra-option learning (macro actions) and policy gradient methods may reduce the effects of slow reacting systems and provide an effective model for continuous valued decisions. The solution to some of the challenge problems will produce several theoretical advances in scaling reinforcement learning methods and establish a standard of excellence, promoting further research on the remaining challenge problems. Furthermore, this research will provide concrete evidence of the long-term contribution of RL-Glue and RL-Library and ensure their usage as the standard benchmarking tool for empirical analysis in the reinforcement learning community for years to come.

Bibliography

[Bagnell, 2004] Bagnell, J. (2004). *Learning Decisions: Robustness, Uncertainty, and Approximation*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

[Boisvert et al., 1997] Boisvert, R. F., Pozo, R., Remington, K., Barrett, R., and Dongarra, J. J. (1997). The Matrix Market: A Web Resource for Test Matrix Collections. In Boisvert, R. F., editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London. Chapman & Hall.

[Boyan and Moore, 1995] Boyan, J. A. and Moore, A. W. (1995). Generalization in Reinforcement Learning: Safely Approximating the Value Function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA. The MIT Press.

[Crites and Barto, 1996] Crites, R. H. and Barto, A. G. (1996). Improving Elevator Performance Using Reinforcement Learning. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1017–1023. The MIT Press.

[Engel et al., 2005] Engel, Y., Mannor, S., and Meir, R. (2005). Reinforcement Learning with Gaussian Processes. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 201–208, New York, NY, USA. ACM Press.

[Hafner, 2005] Hafner, V. V. (2005). Cognitive Maps in Rats and Robots. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 13(2):87–96.

[Langford and Bagnell, 2004] Langford, J. and Bagnell, D. (2004). RLBench, A Reinforcement Learning Benchmark Suite.

[Littman et al., 2005] Littman, M., Riedmillerg, M., and Vlassis, N. (2005). Reinforcement Learning Benchmarks and Bake-offs II: Workshop Proceedings. In *Neural Information Processing Systems*.

[Mahadevan, 1997] Mahadevan, S. (1997). Mountain-Car Problem.

[Montague et al., 1995] Montague, P., Dayan, P., Person, C., and Sejnowski, T. (1995). Bee Foraging in Uncertain Environments Using Predictive Hebbian Learning. *Nature*, 377:725 – 728.

[Moore, 1990] Moore, A. W. (1990). *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge, Cambridge, UK.

[Nevmyvaka et al., 2006] Nevmyvaka, Y., Feng, Y., and Kearns, M. (2006). Reinforcement Learning for Optimized Trade Execution. In *International Conference on Machine Learning*, pages 673 – 680.

[Newman et al., 1998] Newman, D., Hettich, S., Blake, C., and Merz, C. (1998). UCI Repository of Machine Learning Databases.

[Ng et al., 2004] Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2004). Inverted Autonomous Helicopter Flight via Reinforcement Learning. In *International Symposium on Experimental Robotics*.

[Niv et al., 2005] Niv, Y., Duff, M. O., and Dayan, P. (2005). Dopamine, Uncertainty and TD Learning. In *Behavioral and Brain Functions*, volume 6, pages 1744–9081.

[Powers et al., 1995] Powers, K., Sandifer, C., Rice, D., Glick, J., and Cramblitt, B. (1995). Standard Performance Evaluation Corporation.

[Precup et al., 2005] Precup, D., Sutton, R. S., Paduraru, C., Koop, A., and Singh, S. P. (2005). Off-policy Learning with Options and Recognizers. In *Advances in Neural Information Processing Systems 18*, pages 1097–1104.

[Riedmiller, 2005] Riedmiller, M. (2005). Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method. In *European Conference on Machine Learning*, pages 317–328.

[Riedmiller et al., 2003] Riedmiller, M., Lange, S., Timmer, S., and Hafner, R. (2003). CLSquare: Closed Loop Simulation System.

[Simsek and Barto, 2006] Simsek, O. and Barto, A. G. (2006). An Intrinsic Reward Mechanism for Efficient Exploration. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 833–840, New York, NY, USA. ACM Press.

[Singh et al., 2002] Singh, S., Litman, D., and Kearns, M. (2002). Optimizing Dialogue Management with Reinforcement Learning: Experiments with the NJFUN System. *Journal of Artificial Intelligence Research*, 16:105–133.

[Singh and Sutton, 1996] Singh, S. P. and Sutton, R. S. (1996). Reinforcement Learning with Replacing Eligibility Traces. *Machine Learning*, 22(1 - 3):123 – 158.

[Smart and Kaelbling, 2000] Smart, W. D. and Kaelbling, L. P. (2000). Practical Reinforcement Learning in Continuous Spaces. In *Proc. 17th International Conf. on Machine Learning*, pages 903–910. Morgan Kaufmann, San Francisco, CA.

[Stone and Sutton, 2001] Stone, P. and Sutton, R. S. (2001). Scaling Reinforcement Learning Toward RoboCup Soccer. In *Proc. 18th International Conf. on Machine Learning*, pages 537–544. Morgan Kaufmann, San Francisco, CA.

[Sutton, 1996] Sutton, R. S. (1996). Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press.

[Sutton, 2000] Sutton, R. S. (2000). Mountain Car Software.

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

[Sutton et al., 1999] Sutton, R. S., Precup, D., and Singh, S. P. (1999). Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112(1-2):181–211.

[Sutton and Santamaria, 1996] Sutton, R. S. and Santamaria, J. C. (1996). A Standard Interface for Reinforcement Learning Software.

[Szepesvari and Smart, 2004] Szepesvari, C. and Smart, B. (2004). Software.

[Tesauro, 1995] Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Commun. ACM*, 38(3):58–68.

[Wiewiora et al., 2003] Wiewiora, E., Cottrell, G. W., and Elkan, C. (2003). Principled Methods for Advising Reinforcement Learning Agents. In *International Conference on Machine Learning*, pages 792–799.

[Williams, 1992] Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229–256.

[Wingate, 2004] Wingate, D. (2004). Mountain Car.

Appendix A

Task Description Language

The Task_specification is stored as a string with the following format:

$$V : E : O : A$$

V corresponds to the version number of the task description language. If an environment implements new data structures/types that cannot be encoded in the current *Task_ specification* language, then the user may publishe a new description language and increment the version number. E corresponds to the type of task being solved. A character value of 'e' is used for episodic tasks 'c' for continuing tasks. The O and A correspond to observation and action information respectively. The format of O and A are identical. We will describe O only, for brevity.

O contains three components, separated by underscore characters ('_') :

$$\#dimensions\_dimensionTypes\_dimensionRanges$$

The $\#dimensions$ encodes an integer value specifying the number of dimensions in the observation space. The $dimensionTypes$ is a list specifying the type of each dimension variable. The $dimensionTypes$ list is composed of $\#dimensions$ components separated by comma characters (',') and delimited by angle brackets ('[',']'). Each comma-separated value in the list describes the type of values assigned to each observation variable in the environment. In general, observation variables can have one of the following 2 types: 'i' for integer values and 'f' for float values.

The $dimensionRanges$ is a list specifying the range of each dimension variable in the observation space. The $dimensionRanges$ is composed of $\#dimensions$ components

separated by underscore characters and delimited by brackets. Each $dimensionRanges$ component specifies the upper and lower bound of values for each observation variable. An observation space with 2 dimensions would have a $dimensionRanges$ with the following form:

$$[o_1min, o_1max]\_[o_2min, o_2max]$$

The $dimensionRanges$ of an observation space with 1 or more unbounded values cannot be representable in this way. We simply do not specify the range in the $dimensionRanges$ for any observation variables with unbounded values. For example, consider a problem with 3 observation dimensions where the first and third observation variables have interval values and the second has unbounded ratio value. The corresponding $dimensionRanges$ for this problem is encoded as:

$$[o_1min, o_1max]\_[,]\_[o_3min, o_3max]$$

The format of A (action space information) is identical to that of O. The definitions above hold for action spaces.

To provide an illustration of the task specification language, consider the Mountain Car problem. The actions available to the agent are full throttle reverse, zero throttle and full throttle forward. The observation consists of the cars position and velocity. If we encode Actions as 0, 1 and 2 and position and velocity as real values with finite ranges, we get the following Task_specification:

$$"1.2 : e : 2\_[f, f]\_[-1.2, 0.5]\_[-.07, .07] : 1\_[i]\_[0, 2]"$$

This Task_specification provides the following information:

- Task_specification language version 1.2 supported
- task is epsiodic
- observation space has two dimensions
- first observation variable has float values
- second observation variable has float values (2D continuous state)
- range of observation variable one is -1.2 to 0.5

- range of observation variable two is -0.07 to 0.07

- action space has one dimension

- action variable has integer values (discrete actions)

- range of action variable is 0 to 2

Consider a simple gridworld with Actions North, South, East and West and a single dimension observation of grid position. If we encode actions as $\{0, 1, 2, 3\}$ and position as an integer between 0 and $N - 1$, we get the following Task_specification:

$$\text{“}1 : e : 1\_[i]\_[0, N - 1] : 1\_[i]\_[0, 3]\text{”}$$

This Task_specification provides the following information:

- Task_specification language version 1.2 supported

- the task is episodic

- observation space has one dimension

- the observation variable has integer values (discrete state)

- range of the observation variable is 0 to N-1

- the action space has one dimension

- the action variable has integer values (discrete actions)

- range of action variable is 0 to 3