Deep Learning and Reward Design for Reinforcement Learning

by

Xiaoxiao Guo

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Doctoral Committee:

Professor Satinder Singh Baveja, Co-Chair
Professor Richard L. Lewis, Co-Chair
Associate Professor Honglak Lee
Associate Professor Qiaozhu Mei

ACKNOWLEDGEMENTS

First and foremost I would like to thank my co-chairs, Professor Satinder Singh and Professor Rickard Lewis. I am grateful for their continuous guidance and advice in my PhD journey. Their insights and research styles deeply influence and inspire me. I own many thanks to their efforts in training me as an independent researcher. This dissertation could not happen without their patience and support.

I would like to thank all my doctoral committee members. Professor Honglak Lee is an amazing person to work with. He is hands-on and knowledgeable about the practice of machine learning, especially deep learning. Professor Qiaozhu Mei introduces me to a broader scope of machine learning applications, and he is always willing to give invaluable advice on being an international student in the US.

I also want to thank my lab-mates, Jeshua Bratman, Rob Cohen, Michael Shvartsman, Nan Jiang, Ananda Narayan, Junhyuk Oh, Shun Zhang, Qi Zhang, and Janarthanan Rajendran, for their mentorship and friendship. I have learned a lot from our enlightening discussions, and this dissertation has benefited from the conversations with them.

Most of all, thank Cheng Li for achieving this with me.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Deep Learning and Reward Design for Reinforcement Learning

by

Xiaoxiao Guo

Co-Chairs: Satinder Singh Baveja and Richard L. Lewis

One of the fundamental problems in Artificial Intelligence is sequential decision making in a flexible environment. Reinforcement Learning (RL) gives a set of tools for solving sequential decision problems. Although the theory of RL addresses a general class of learning problems with a constructive mathematical formulation, the challenges posed by the interaction of rich perception and delayed rewards in many domains remain a significant barrier to the widespread applicability of RL methods.

The rich perception problem itself has two components: 1) the sensors at any time step do not capture all the information in the history of observations, leading to partial observability, and 2) the sensors provide very high-dimensional observations, such as images and natural languages, that introduce computational and sample-complexity challenges for the representation and generalization problems in policy selection. The delayed reward problem—that the effect of actions in terms of future rewards is delayed in time—makes it hard to determine how to credit action sequences for reward outcomes.

This dissertation offers a set of contributions that adapt the hierarchical representation learning power of deep learning to address rich perception in vision and text domains, and develop new reward design algorithms to address delayed rewards. The first contribution is a new learning method for deep neural networks in vision-based real-time control. The learning method distills slow policies of the Monte Carlo Tree Search (MCTS) into fast convolutional neural networks, which outperforms the conventional Deep Q-Network. The second contribution is a new end-to-end reward design algorithm to mitigate the delayed rewards for the state-of-the-art MCTS method. The reward design algorithm converts visual perceptions into reward bonuses via deep neural networks, and optimizes the network weights to improve the performance of MCTS end-to-end via policy gradient. The third contribution is to extend existing policy gradient reward design method from single task to multiple tasks. Reward bonuses learned from old tasks are transferred to new tasks to facilitate learning. The final contribution is an application of deep reinforcement learning to another type of rich perception, ambiguous texts. A synthetic data set is proposed to evaluate the querying, reasoning and question-answering abilities of RL agents, and a deep memory network architecture is applied to solve these challenging problems to substantial degrees.

CHAPTER I

Introduction

1.1 Challenges in Reinforcement Learning

One of the fundamental problems in Artificial Intelligence is sequential decision making in a stochastic environment. As a motivating example, consider the problem of designing a robot that can learn to play a variety of video games automatically. The game playing robot is in a stochastic environment because the games exhibit random, unpredictable events / consequences. The design of game playing robot also represents a sequential decision-making problem, in that the robot's utility depends on a sequence of decisions. The utility of each decision is usually not known and can only be learned via trial-and-errors. Sequential decision problems incorporate utilities, uncertainty, and sensing, and include search and planning problems as special cases.

The sequential decision making problem in a stochastic environment is formalized using the Markov Decision Process (MDP) framework. Briefly, it models the stochastic environment as being in some states at each time step. For example, a state of the Chess game is the game board. The agent selects actions to interact with the environment. As a result of the action selection, the environment moves through some sequences of states, and the agent receives bounded scalar signals, i.e. rewards, at each time step as immediate feedbacks. The task to solve a sequential decision problem is to derive a policy that maps states to actions in order to maximize the expected sum of the received rewards. To sum up, a sequential decision problem for a fully observable, stochastic environment with a Markovian state transition model and additive rewards is called a Markov Decision Process. A large and diverse set of problems can be modeled using the MDP framework, such as robot planning and navigation, dialog systems, power supply management, human-computer interaction design and digital marketing.

Reinforcement Learning (RL) gives a set of tools for solving MDP problems. RL agents learn to act over time in a stochastic environment to achieve high cumulative rewards. Historically, RL methods focus on two unique-to-RL problems: (1) the agents need to take actions that seem sub-optimal based on current knowledge to allow for the possibility that new knowledge obtained by taking them may yield even higher reward in the future (exploration versus exploitation), and (2) the effect of actions in terms of future rewards is delayed in time and this makes it hard to determine which actions in a sequence are responsible (delayed rewards or temporal credit assignment). Traditional RL methods have numerous successes on solving this two unique problems. Although the theory of RL addresses a general class of learning problems with a constructive mathematical formulation, the challenges posed by the interaction of rich perception and delayed rewards in many domains remain a significant barrier to the widespread applicability of RL methods.

1.1.1 Deep Learning and Challenges of Rich Perception

Many real-world RL problems combine the challenges of closed loop action (or policy) selection with the already significant challenges of high-dimensional perception (shared with many Supervised Learning problems). RL has made substantial progress on theory and algorithms for policy selection (the distinguishing problem of RL), but these contributions have not directly addressed problems of perception. The perception problem itself has two components: 1) the sensors at any time step do not capture all the information in the history of observations, leading to partial observability, and 2) the sensors provide very high-dimensional observations, such as images and natural languages, that introduce computational and sample-complexity challenges for the representation and generalization problems in policy selection.

One way to handle the perception challenges when a model of the RL environment is available is to avoid the perception problem entirely by eschewing the building of an explicit policy and instead using repeated incremental planning via Monte Carlo Tree Search (MCTS) methods, such as UCT (*Kocsis and Szepesvári* (2006)). UCT uses the state transition model to simulate look-ahead roll-out trajectories, and it returns the Monte-Carlo average of the discounted sum of rewards over the sampled trajectories for each action at the root node as its estimate of the utility of taking that action in the current state. UCT has the nice theoretical property that the number of simulation steps needed to ensure any bound on the loss of following the UCT policy is independent of the size of the state space. The results capture the fact that the use of MCTS avoids the perception problem, but at the cost of requiring substantial computation for every time step of action selection because it never builds an explicit policy.

Either when a model is not available, or when an explicit representation of the policy is required, the usual approach to applied RL success has been to use expert-developed task-specific features of a short history of observations in combination with function approximation methods. However, these expert-developed task-specific features are not very practical because of at least three reasons. First, the feature representations often have to be obtained by human efforts. The domain-experts need to spend a significant amount of time on hand-tuning the features based on their knowledge and numerous trials and errors. Second, the features in one domain usually do not generalize to other domains. It means that the tedious, effort-consuming hand-tuning processes need to be repeated every time when a new task arises. Third, the hand-crafted features are not even feasible when there is little knowledge of the task or the underlying decision problem is too complex. Eliminating the dependence of applied RL success on engineered features is one of the key challenges for RL with rich perception.

In the last decade, Deep Learning (DL) approaches have made remarkable progress on the perception problem. Deep learning (see *Bengio* (2009); *Schmidhuber* (2015) for a survey) has emerged as a powerful technique for *learning* feature representations from data (again, this is in contrast to the usual approach of hand-crafting of features by domain experts). For example, DL has achieved state-of-the-art results in image classification (*Krizhevsky et al.* (2012); *Szegedy et al.* (2015)), speech recognition (*Graves et al.* (2013); *Hannun et al.* (2014)), and activity recognition (*Karpathy et al.* (2014)). In DL, features are learned in a compositional hierarchy. Specifically, low-level features are learned to encode low-level statistical dependencies (e.g., "edges" in images), and higher-level features encode higher-order dependencies of the lower-level features (e.g., "object parts") (*Lee et al.* (2009)). RL and DL share the aim of generality, in that they both intend to minimize or eliminate domain-specific engineering, while providing off-the-shelf performance that competes with or exceeds systems that exploit control heuristics and hand-coded features. Combining modern RL and DL approaches therefore offers the potential solutions for RL problems with rich perception.

The highly influential work in combining RL and DL, Deep Q-Network (DQN), uses a Convolutional Neural Network (CNN) for state representation learning and function approximation. DQN samples a huge amount of self-playing data to learn the action values via improved Neural Fitted Q-Iteration updates. DQN achieves a breakthrough on complex RL problems with rich perception, the Arcade Learning Environment benchmark (ATARI games). This line of work in combining DL and RL has yielded remarkable empirical successes for reinforcement learning tasks in recent years. Unlike most of the work that combines model-free RL methods and Deep Learning architectures, my contributions focus on combining simulator-based RL methods (i.e., Monte Carlo Tree Search) and Deep Learning. Chapter II, III and V of the dissertation summarize my efforts in combining DL and RL for vision-based real-time control and text-based reasoning tasks.

1.1.2 Reward Design and Challenges of Delayed Rewards

There has also been considerable emphasis (and associated progress) on algorithm and theory development for dealing with delayed reward (*Sutton and Barto* (1998); *Szepesvári* (2010)). Among RL methods, Monte Carlo Tree Search (MCTS) methods have shown extremely powerful for complex reinforcement learning problems. For example, the strongest RL programs for most classic games are based on MCTS methods, such as AlphaGo for Go (*Silver et al.* (2016)), Meep for Chess (*Veness et al.* (2009)), Logistello for Othello (*Buro* (2002)), Maven for Scrabble (*Sheppard* (2002)), and Chinook for Checkers (*Schaeffer et al.* (1992)). MCTS methods use Monte Carlo roll-out trajectories to estimate the state or state-action values in order to solve the delayed rewards in reinforcement learning tasks. As more trajectories are executed, the search tree grows deeper and larger, and the estimate values become more stable and accurate. The search policy to determine action selection during tree construction is also improved over time by selecting actions with higher estimate values. Given sufficient search trajectories, the search policy converges to the optimal policy, and the estimate values converge to the optimal values. However, the computation required by MCTS to converge to an optimal policy for a complex RL domain is usually huge and depends on the delay of rewards. MCTS methods suffer when the sampling trajectories are not sufficient with respect to the delay of rewards.

Researchers have developed several principles to mitigate the computation limits of MCTS methods. One method to reduce the depth of tree search is the classical leaf-evaluation. The leaf-evaluation approaches truncate the search tree at a leaf state and replace the subtree below it by an approximation value. The value to compensate for the missing subtree below the leaf is often designed using domain heuristics or learned from previous planning trajectories or real execution experience. The breadth of the search tree may also be reduced by sampling mostly on the promising actions. Averaging over these roll-out trajectories thus provides an effective value estimation. Many other approaches improve MCTS by mechanisms for generalization of returns and visit statistics across states: e.g., the transposition tables of *Childs et al.* (2008), the rapid action value estimation of *Gelly and Silver* (2011), the local homomorphisms of *Jiang et al.* (2014), the state abstractions of *Hostetler et al.* (2014), and the local manifolds of *Srinivasan et al.* (2015).

This dissertation focuses on reward bonus learning methods to improve MCTS methods. Unlike other approaches, the reward bonus design approaches learn the bonus directly from the MCTS planners' execution trajectories, without auxiliary training signals (i.e., reward prediction error). In almost all of RL research, the reward function is considered part of the task specification and thus unchangeable. The optimal reward framework of *Singh et al.* (2010) stems from the observation that a reward function plays two roles simultaneously in RL problems. The first role is that of *evaluation* in that the task-specifying reward function is used by the agent designer to evaluate the actual behavior of the agent. The second is that of *guidance* in that the reward function is also used by the RL algorithm implemented by the agent to determine its behavior (e.g., via Q-learning (*Watkins and Dayan* (1992)) or UCT planning. The optimal reward problem separates these two roles into two separate reward functions, the task-specifying *objective* reward function used to evaluate performance, and an *internal* reward function used to guide agent behavior. Given a Controlled Markov Process (CMP) M with an objective reward function R^o, an agent $\mathcal{A}$ parameterized by an internal reward function, and a space of possible internal reward functions $\mathcal{R}$, an optimal internal reward function R^{i^*} is defined as follows:

$$R^{i^*} = arg \max_{R^i \in \mathcal{R}} \mathbb{E}_{h \sim \langle \mathcal{A}(R^i), M \rangle} \Big\{ U^o(h) \Big\},$$

where $\mathcal{A}(R^i)$ is the agent with internal reward function R^i, $h \sim \langle \mathcal{A}(R^i), M \rangle$ is a random history (trajectory of alternating states and actions) obtained by the interaction of agent $\mathcal{A}(R^i)$ with CMP M, and $U^o(h)$ is the objective utility (as specified by R^o) of interaction history h. The optimal internal reward function will depend on the agent $\mathcal{A}$'s architecture and its limitations, and this distinguishes ORP from other reward-design approaches such as inverse-RL. When would the optimal internal reward function be different from the objective reward function? If an agent is *unbounded* in its capabilities with respect to the CMP then the objective reward function is always an optimal internal reward function. More crucially though, in the realistic setting of *bounded* agents, optimal internal reward functions may be quite different from objective reward functions. *Singh et al.* (2010) and *Sorg et al.* (2010a) provide many examples and some theory of when a good choice of internal reward can mitigate agent bounds, including bounds corresponding to limited lifetime to learn, limited memory, and limited resources for planning.

Computing R^{i^*} can be computationally non-trivial. *Sorg et al.* (2010a) propose policy gradient reward design (PGRD) method based on the insight that any planning algorithm can be viewed as procedurally translating the internal reward function R^i into behavior—that is, R^i are indirect parameters of the agent's policy. However, previous applications of reward design have been limited in at least two ways: First, they have required careful hand-construction of features that define a reward-bonus function space to be searched over, and they have limited the reward-bonus function space to be a linear function of hand-constructed features. As discussed earlier, eliminating the dependence of applied RL success on engineered features is one of the key challenges for RL applications. Deep Learning is an effective replacement for hand-crafted features. Moreover, deep neural networks provide trainable non-linear reward functions that previous reward design methods lack. Chapter III explores combining reward design and DL to make adapting rewards for MCTS methods for more widely and practically applicable.

A second limitation of previous reward design work is that they only focus on single task scenarios. Although RL research results in a rich set of algorithms for improving task performance with experience, how to effectively transfer learned skills and knowledge from one problem setting to another is still an open question. Be more specific, consider a RL agent solving a sequence of n tasks, M_1, M_2, ..., M_n. After the agent has learned to solve tasks M_1, M_2, ..., M_{n-1}, the agent then learns to solve the **related but different** task, M_n. It is intuitively reasonable that the agent would be able to learn the task M_n faster than from scratch via reusing the knowledge gained in the learned solutions for M_1, M_2, ..., M_{n-1}. The idea behind transfer in RL seems intuitively clear. The nature of the task relationship will define how transfer can take place. One contribution of this dissertation is to develop new algorithm to transfer reward bonuses in task sequences where all of the tasks have the same state space and action set, but different state transition probabilities and reward functions.

1.2 Summary of this dissertation

This dissertation offers a set of contributions that adapt the hierarchical representation learning power of deep learning to address rich perception in vision and text domains, and develop new reward design algorithms to address delayed rewards. A summary of the contributions of this dissertation is shown in Figure 1.1. Most of the contributions described in this thesis have first appeared as various publications: Chapter II (*Guo et al.* (2014)), Chapter III (*Guo et al.* (2016)), Chapter IV (*Guo et al.* (2013)). The first contribution is a new learning method for deep neural networks in vision-based real-time control. The learning method distills slow policies of

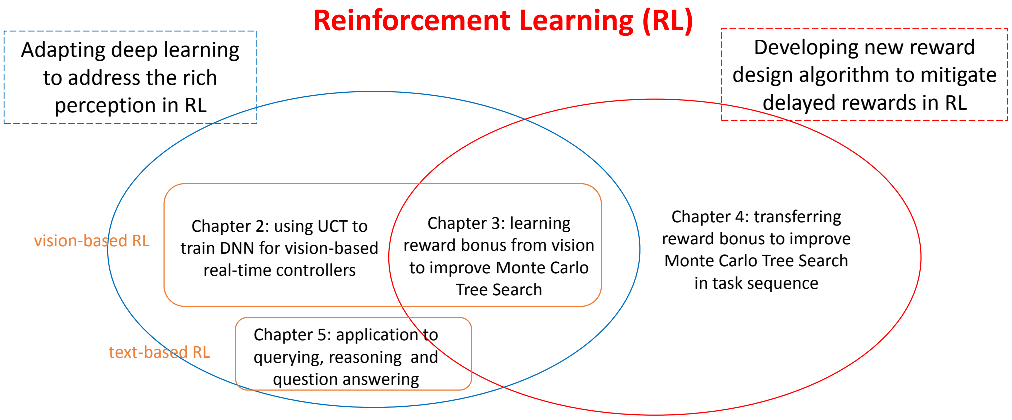

Reinforcement Learning (RL)
Adapting deep learning to address the rich perception in RL
Developing new reward design algorithm to mitigate delayed rewards in RL
vision-based RL
text-based RL
Chapter 2: using UCT to train DNN for vision-based real-time controllers
Chapter 3: learning reward bonus from vision to improve Monte Carlo Tree Search
Chapter 4: transferring reward bonus to improve Monte Carlo Tree Search in task sequence
Chapter 5: application to querying, reasoning and question answering

Figure 1.1: The road-map of this dissertation.

the Monte Carlo Tree Search (MCTS) into fast convolutional neural networks, which outperforms the conventional Deep Q-Network. The second contribution is a new end-to-end reward design algorithm to mitigate the delayed rewards for the state-of-the-art MCTS method. The reward design algorithm converts visual perceptions into reward bonuses via deep neural networks, and optimizes the network weights to improve the performance of MCTS end-to-end via policy gradient. The third contribution is to extend existing policy gradient reward design method from single tasks to multiple tasks. Reward bonuses learned from old tasks are transferred to new tasks to facilitate learning. The final contribution is an application of deep reinforcement learning to another type of rich perception, ambiguous texts. A synthetic data set is proposed to evaluate the querying, reasoning and question-answering abilities, and a deep memory network architecture is applied to solve these challenging problems to substantial degrees.

1.2.1 Deep Learning for Real-time Control using Offline Monte Carlo Tree Search

Conventional deep reinforcement learning methods, such as Deep Q-Network (DQN), require a huge amount of simulated data to learn good policies. Those simulation data could be instead organized by Monte Carlo Tree Search (MCTS) methods to construct even better policies. However, MCTS methods do not have explicit policies and need to do repeated look-ahead tree building at every decision time. Usually, the look-ahead tree-building procedure is time-consuming and memory demanding. These drawbacks of MCTS methods make them incapable of real-time control. Chapter II explores the combination of the well-performing policies of slow MCTS methods and the fast decision making capability of deep neural networks for vision-based real-time control tasks. The contribution is a new learning method for deep neural networks in vision-based real-time control. The learning method distills slow policies of the Monte Carlo Tree Search (MCTS) into fast convolutional neural networks, which outperforms the conventional Deep Q-Network. Different training objectives are compared, and empirical results show that policy-based signals result in better-performing neural networks than value-based signals.

1.2.2 Deep Learning for Reward Design to Improve Monte Carlo Tree Search

Chapter III offers a novel means of combining DL and RL. There has been a flurry of recent work on combining DL and RL, including the seminal work using DL as a function approximator for Q-learning (*Mnih et al.* (2015)), the use of UCT-based planning to provide policy-training data for a DL function approximator (*Guo et al.* (2014)), the use of DL to learn transition-models of ATARI games to improve exploration in Q-learning (*Oh et al.* (2015); *Stadie et al.* (2015)), and the use of DL as a parametric representation of policies to improve via policy-gradient approaches (*Schulman et al.* (2015)). Chapter III uses DL as a function approximator to learn reward-bonus functions from experience to mitigate computational limitations in UCT (*Kocsis and Szepesvári* (2006)), a Monte Carlo Tree Search (MCTS) algorithm. The work builds on PGRD (policy-gradient for reward-design; *Sorg et al.*

(2010a)), a method for learning reward-bonus functions for use in planning. Previous applications of PGRD have been limited in a few ways: 1) they have been applied to small RL problems; 2) they have required careful hand-construction of features that define a reward-bonus function space to be searched over by PGRD; 3) they have limited the reward-bonus function space to be a linear function of hand-constructed features; and 4) they have high-variance in the gradient estimates (this issue was not apparent in earlier work because of the small size of the domains). Chapter III addresses all of these limitations by developing PGRD-DL, a PGRD variant that automatically learns features over raw perception inputs via a multi-layer convolutional neural network (CNN), and uses a variance-reducing form of gradient. The proposed new algorithm is tested on the benchmark for deep reinforcement learning (the Arcade Learning Environment). We show that PGRD-DL can improve performance of UCT on multiple ATARI games.

1.2.3 Reward Transfer for Sequence Multiple Tasks

Chapter IV extends the reward design approach to mitigate delayed rewards from single task scenarios to sequential multiple tasks. Previous transfer learning methods in RL focus on learning setting, in which the agent is assumed to lack complete knowledge of the task, i.e. either the state transition model or the reward function is not given. In the learning setting, transfer in RL has explored the reuse across tasks of many different components of a RL agent, including value functions (*Tanaka and Yamamura* (2003); *Konidaris and Barto* (2006); *Liu and Stone* (2006)), policies (*Natarajan and Tadepalli* (2005); *Torrey and Shavlik* (2010)), and models of the environment (*Atkeson and Santamaria* (1997); *Taylor et al.* (2008)). Other transfer approaches have considered parameter transfer (*Taylor et al.* (2007)), selective reuse of sample trajectories from previous tasks (*Lazaric et al.* (2008)), as well as reuse of learned abstract representations such as state abstraction and options (*Perkins* *and Precup* (1999); *Konidaris and Barto* (2007)). In contrast, Chapter IV considers the knowledge transfer problem in the planning setting, or being more specific, the intertask knowledge generalization for UCT planners.

In the planning setting, the RL agent is assumed to know the MDPs perfectly as well as their changes. If the planning agent were unbounded in planning capacity (practically impossible), there would be nothing interesting left to consider because the planning agent could simply find the optimal policy for each new task and execute it. What makes the intertask generalization problem interesting therefore is that the UCT-based planning agent is computationally limited: the depth and the number of trajectories feasible are small enough that it cannot find the optimal policies. Note that basic UCT does use a reward function but does not use an initial value function or policy and hence changing and reusing a reward function is a natural and consequential way to influence UCT. While non-trivial modifications of UCT could allow use of value functions and/or policies, we do not consider them here. (Actually, reward functions strictly subsume the leaf-evaluation value functions for UCT, i.e., there exists a reward function for every leaf-evaluation heuristic that leads to equivalent behavior, but the converse is not true. The learning of reward function can be applied to leaf-evaluation function directly as well.) In addition, in the planning setting, the state model is available to the agent and so there is no scope for transfer by reuse of model knowledge. Thus, the reuse of reward functions may well be the most consequential option available in UCT. The main contribution of Chapter IV is the idea to learn a reward mapping function that maps task parameters to reward bonus parameters to transfer reward bonuses for UCT planners in task sequence.

1.2.4 Learning to Query, Reason, and Answer Questions on Ambiguous Texts

Unlike the work in vision-based RL domains in Chapter II and Chapter III, Chapter V applies DL on text-based dialog management tasks to address a different type of rich perception in RL. A key goal of research in conversational systems is to train an interactive agent to help a user with a task. Human conversation, however, is notoriously incomplete, ambiguous, and full of extraneous detail. To operate effectively, the agent must not only understand what was explicitly conveyed but also be able to reason in the presence of missing or unclear information. When unable to resolve ambiguities on its own, the agent must be able to ask the user for the necessary clarifications and incorporate the response in its reasoning. Motivated by this problem, Chapter V introduces QRAQ ("crack"; Query, Reason, and Answer Questions), a new synthetic domain, in which a user gives an agent a short story and asks a challenge question. These problems are designed to test the reasoning and interaction capabilities of a learning-based agent in a setting that requires multiple conversational turns. A good agent should ask only non-deducible, relevant questions until it has enough information to correctly answer the user's question. Chapter V explores standard and improved reinforcement learning based memory-network architectures to solve QRAQ problems in the real-world application setting where the reward signal can only tell the agent if its final answer to the challenge question is correct or not. Chapter V evaluates the proposed architectures on four QRAQ dataset types, and scales the complexity for each along multiple dimensions. The main applied results show that both architectures solve the challenging problems to substantial degrees.

CHAPTER II

Deep Learning for Real-time Control Using Offline Monte Carlo Tree Search

Many real-world Reinforcement Learning (RL) problems combine the challenges of closed-loop action (or policy) selection with the already significant challenges of high-dimensional perception (shared with many Supervised Learning problems). RL has made substantial progress on theory and algorithms for policy selection (the distinguishing problem of RL), but these contributions have not directly addressed problems of perception. Deep learning (DL) approaches have made remarkable progress on the perception problem but do not directly address policy selection. RL and DL methods share the aim of generality, in that they both intend to minimize or eliminate domain-specific engineering, while providing "off-the-shelf" performance that competes with or exceeds systems that exploit control heuristics and hand-coded features. Combining modern RL and DL approaches therefore offers the potential for general methods that address challenging applications requiring both rich perception and policy-selection.

The Arcade Learning Environment (ALE) is a relatively new and widely accessible class of benchmark RL problems(*Bellemare et al.* (2013b)) that provide a particularly challenging combination of policy selection and perception. ALE includes an emulator and about 50 Atari 2600 (a 1970s–80s home-video console) games. The complexity and diversity of the games—both in terms of perceptual challenges in mapping pixels to useful features for control and in terms of the control policies needed—make ALE a useful set of benchmark RL problems, especially for evaluating general methods intended to achieve success without hand-engineered features.

Since the introduction of ALE, there have been a number of attempts to build general-purpose Atari game playing agents. The departure point for this chapter is a highly influential and significant breakthrough (*Mnih et al.* (2013)) that combines RL and DL to build agents for multiple Atari games. It achieved the best machine-agent real-time game play when it was published (in some games close to or better than human-level play). It does not require feature engineering, and indeed reuses the same perception architecture and RL algorithm across all the games. We believe that continued progress on the ALE environment that preserves these advantages will extend to broad advances in other domains with significant perception and policy selection challenges. Thus, our immediate goal in the work reported here is to build even better performing general-purpose Atari game playing agents. We achieve this by introducing new methods for combining RL and DL that use slow, off-line Monte Carlo tree search planning methods to generate training data for a deep-learned classifier capable of state-of-the-art real-time play.

2.1 Related Work

Arcade Learning Environment. The combination of modern Reinforcement Learning and Deep Learning approaches holds the promise of making significant progress on challenging applications requiring both rich perception and policy-selection. The Arcade Learning Environment (ALE) provides a set of Atari games that represent a useful benchmark set of such applications. Figure 2.1 shows 12 representative games to visually illustrate their conceptual variety (see caption for details). While the games in ALE are simpler than many modern games, they still pose significant chal-

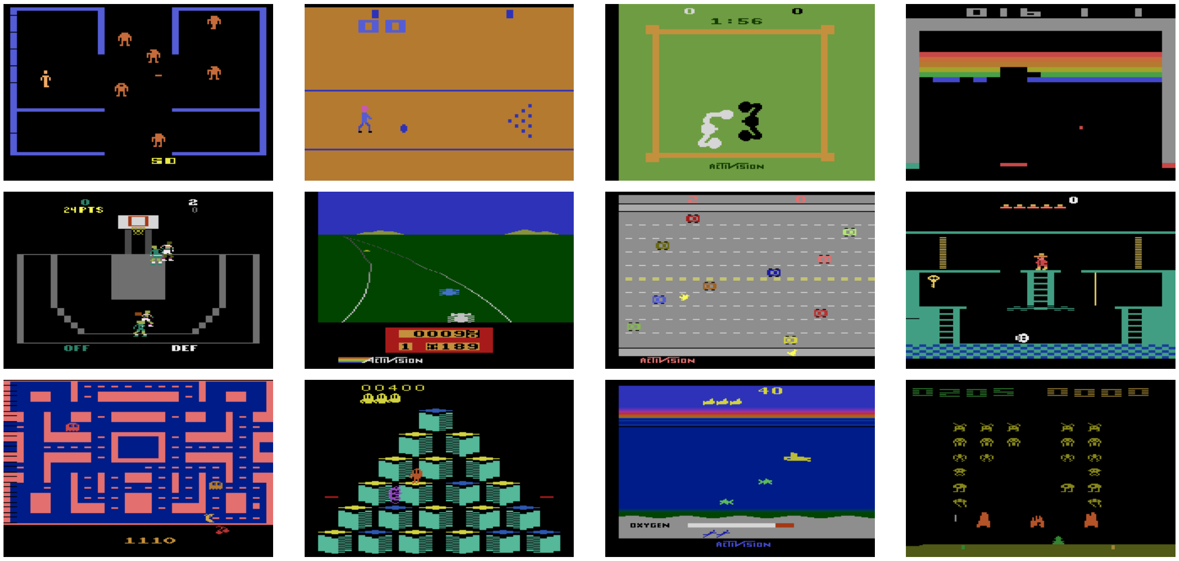


Figure 2.1: Illustration of the variety of games within ALE (from left to right and top to bottom). *Berzerk*: Control an avatar that uses a laser-like weapon to shoot robot enemies in a maze of randomly generated rooms. *Bowling*: A simulation of the sport of bowling. *Boxing*: the game shows a top-down view of two boxers. The player's avatar and the opponent can hit each other with a punch when close enough. *Breakout*: Use the paddle to bounce the ball and destroy all the bricks. *Double Dunk*: A simulation of two-on-two, half-court basketball. *Enduro*: Drive a car in a race to pass a certain number of cars each day in order to continue the race the next simulated day. *Freeway*: Control chickens to run across a ten lane highway filled with traffic. *Montezumma's Revenge*: Move an avatar from room to room in a dangerous labyrinthine, gathering jewels and killing enemies along the way. *Ms. Pac-Man*: Move the agent to traverse a maze, consuming all the wafers while avoiding four ghosts. *Q-bert*: Change the color of every cube in a pyramid by making the avatar hop on top of the cube while avoiding obstacles and enemies. *Seaquest*: Control a submarine to avoid or destroy various objects at different levels. The player also needs to pick up divers under water and get air from the surface. *Space Invaders*: Defeat waves of aliens with a laser cannon.

lenges to human players. In RL terms, for a human player these games are Partially-Observable Markov Decision Processes (POMDPs). The true state of each game at any given point is captured by the contents of the limited random-access memory (RAM). A human player does not observe the state and instead perceives the game screen (frame) which is a 2D array of 7-bit pixels, 160 pixels wide by 210 pixels high.

The action space available to the player depends on the game but maximally consists of the 18 discrete actions defined by the joystick controller. The next state is a deterministic function of the previous state and the player's action choice. Any stochasticity in these games is limited to the choice of the initial state of the game (which can include a random number seed stored in RAM). So even though the state transitions are deterministic, the transitions from history of observations and actions to next observation can be stochastic (because of the stochastic initial hidden state). The immediate reward at any given step is defined by the game and made available by the ALE; it is usually a function of the current frame or the difference between current and previous frames. When running in real-time, the simulator generates 60 frames per second. In the maximum speed configuration, the simulator can speed up to about 6,000 frames per second. All the games we consider terminate in a finite number of time-steps (and so are episodic). The goal in these games is to select an optimal policy, i.e., to select actions in such a way so as to maximize the expected value of the cumulative sum of rewards until termination.

From the point of view of game-playing policies, there are general subtasks/subgoals shared across subsets of the games. We describe here some of this common task structure, along with how variation in this task structure crucially gives rise to variation in the degree of delayed reward and partial observability.

Eliminating objects and avoiding collision. In many games, the agent can score small amounts of reward by eliminating objects by shooting at them or by just avoiding collision with specific objects. Because such reward, when available, is often only delayed by tens of steps, it is relatively easy to learn to solve this kind of subtasks.

Approaching objects or locations. In some domains, the agent has to obtain objects by approaching them. In other cases, the agent has to approach goal locations to move to a different game level. If immediate reward is obtained upon approaching such locations or upon capturing such objects, then those subtasks might be easy to

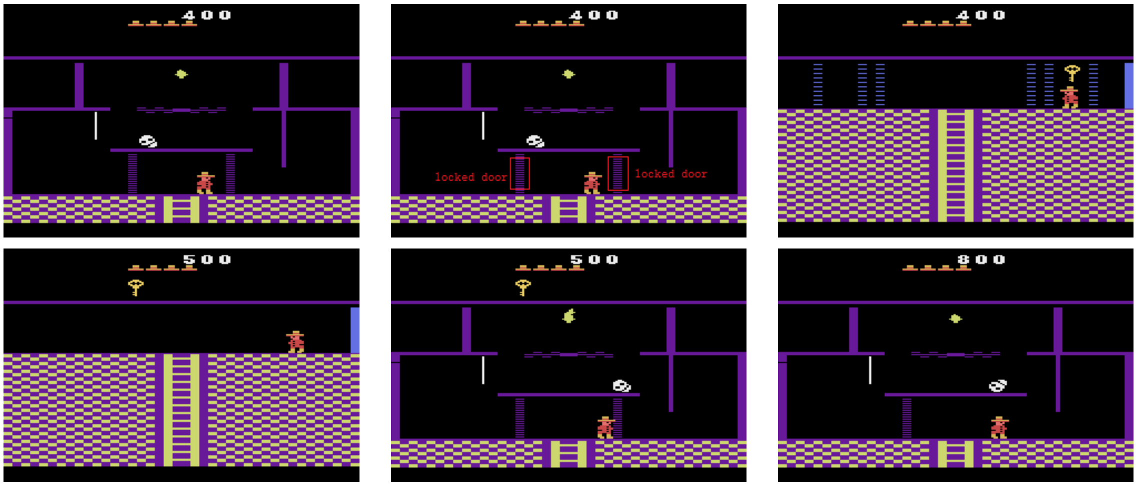


Figure 2.2: A sequence of frames from human play providing evidence of deep partial observability in *Montezuma's Revenge*. The top row shows the game agent encountering locked doors, shown highlighted in the top-middle frame, and then the top-right frame after hundreds of steps in which the player finds a key in a completely different space with no overlap with the earlier space containing the locked doors. The bottom row shows that the human player successfully brings the game agent back to the space with the locked doors, goes to the door (bottom-middle), and then exits the room consuming the key (bottom-right).

learn. If these objects don't directly give reward but can be used subsequently as a tool to obtain reward after hundreds or thousands of steps, this highly delayed reward makes such subtasks very difficult.

Returning to locations or objects. In many games, the agent has to remember to go back to some location from the past that is no longer on the screen. This can lead to a very high degree of partial observability and we show an example in Figure 2.2 from the game of *Montezuma's Revenge*.

Preserving lives and other resources. In most games, the player is given a small number (> 1) of lives. To human players, it is obvious that they should preserve lives, but to the machine agent losing a life does not provide any immediate negative reward. The effect is in the very long run a reduced ability to accumulate future reward. A similar effect is true for limited resources such as shields or weapons that may be useful to preserve. The very long delay in the consequences of resource loss makes this a significant challenge for agents to learn.

In summary, our preliminary analysis of a dozen or so games shows that they vary widely in the delay in reward (from tens of steps in *Beam Rider* and *Enduro* to low hundreds of steps in *Breakout* and *Freeway*, to low thousands in *Seaquest*, and even high thousands as in *Montezuma's Revenge*), as well as in the degree of partial observability from *Breakout* and *Freeway* being second-order Markov to *Montezuma's Revenge* being several-thousands-order Markov.

Challenges of Rich Perception in RL. RL and more broadly decision-theoretic planning have a suite of methods that address the challenge of selecting/learning good policies, including value function approximation, policy search, and Monte-Carlo Tree Search (*Kearns et al.* (2002); *Kocsis and Szepesvári* (2006)) (MCTS). These methods have different strengths and weaknesses, and there is increasing understanding of how to match them to different types of RL-environments. Indeed, an accumulating number of applications attest to this success. But it is still not the case that there are reasonably off-the-shelf approaches to solving complex RL problems of interest to Artificial Intelligence, such as the games in ALE. One reason for this is that despite major advances there hasn't been an off-the-shelf approach to significant perception problems. The perception problem itself has two components: 1) the sensors at any time step do not capture all the information in the history of observations, leading to partial observability, and 2) the sensors provide very high-dimensional observations that introduce computational and sample-complexity challenges for policy selection.

One way to handle the perception challenges when a model of the RL environment is available is to avoid the perception problem entirely by eschewing the building of an explicit policy and instead using repeated incremental planning via MCTS methods such as UCT (*Kocsis and Szepesvári* (2006)) (discussed below). Either when a model is not available, or when an explicit representation of the policy is required, the usual approach to applied RL success has been to use expert-developed task-specific features of a short history of observations in combination with function approximation methods and some trial-and-error on the part of the application developer (on small enough problems this can be augmented with some automated feature selection methods). Eliminating the dependence of applied RL success on engineered features motivates our interest in combining RL and DL (though see *Tesauro* (1995) for an example of early work in this direction).

Over the past decades, deep learning (see *Bengio* (2009); *Schmidhuber* (2015) for a survey) has emerged as a powerful technique for *learning* feature representations from data (again, this is in contrast to the usual approach of hand-crafting of features by domain experts). For example, DL has achieved state-of-the-art results in image classification (*Krizhevsky et al.* (2012); *Ciresan et al.* (2012)), speech recognition (*Mohamed et al.* (2012); *Graves et al.* (2013)), and activity recognition (*Le et al.* (2011); *Karpathy et al.* (2014)). In DL, features are learned in a compositional hierarchy. Specifically, low-level features are learned to encode low-level statistical dependencies (e.g., "edges" in images), and higher-level features encode higher-order dependencies of the lower-level features (e.g., "object parts") (*Lee et al.* (2009)). In particular, for data that has strong spatial or temporal dependencies, convolutional neural networks (*LeCun et al.* (1998)) have been shown to learn invariant high-level features that are informative for supervised tasks. Such convolutional neural networks were used in the recent successful combination of DL and RL for Atari Game playing (*Mnih et al.* (2013)) that forms the departure point of our work.

Model-Free RL Agents for Atari Games. Here we discuss work that does **not** access the state in the games and thus solves the game as a POMDP. In principle, one could learn a state representation and infer an associated MDP model using frame-observation and action trajectories, but these games are so complex that this is rarely done (though see *Bellemare et al.* (2014) for a recent attempt). Instead, partial observability is dealt with by hand-engineering features of short histories of frames observed so far and model-free RL methods are used to learn good policies as a function of those feature representations. For example, the paper that introduced ALE (*Bellemare et al.* (2013b)), used SARSA with several different hand-engineered feature sets. The contingency awareness approach (*Bellemare et al.* (2012b)) improved performance of the SARSA algorithm by augmenting the feature sets with a learned representation of the parts of the screen that are under the agent's control. The sketch-based approach (*Bellemare et al.* (2012a)) further improves performance by using the tug-of-war sketch features. HyperNEAT-GGP *Hausknecht et al.* (2012)) introduces an evolutionary policy search based Atari game player. Most recently Deep Q-Network (hereafter DQN) (*Mnih et al.* (2013)) uses a modified version of Q-Learning with a convolutional neural network (CNN) with three hidden layers for function approximation. This last approach was the state of the art in this class of methods for Atari games before the work of this chapter was done and is the basis for our work[1]; we present the relevant details in Section 2.3. It does not use hand-engineered features but instead provides the last four raw frames as input (four instead of one to alleviate partial observability).

Planning Agents for Atari Games Based on UCT. These approaches access the state of the game from the emulator and hence face a deterministic MDP (other than the random choice of initial state). They incrementally plan the action to take in the current state using UCT, an algorithm widely used for games. UCT has three parameters, the number of trajectories, the maximum-depth (uniform for each trajectory), and an exploration parameter (a scalar set to 1 in all our experiments). In general, the larger the trajectory & depth parameters are, the slower UCT is but the better it is. UCT uses the emulator as a model to simulate trajectories as follows. Suppose it is generating the k^{th} trajectory and the current node is at

[1] The performance of DQN is improved in later work by using deeper neural networks (*Mnih et al.* (2015)), and adapting more robust update rules (*van Hasselt et al.* (2016)).

depth d and the current state is s. It computes a score for each possible action a in state-depth pair (s, d) as the sum of two terms, an exploitation term that is the Monte-Carlo average of the discounted sum of rewards obtained from experiences with state-depth pair (s, d) in the previous $k - 1$ trajectories, and an exploration term that is $\sqrt{\log\left(n(s,d)\right)/n(s,a,d)}$ where $n(s, d)$ and $n(s, a, d)$ are the number of simulated-experiences of state-depth pair (s, d), and of action a in state-depth pair (s, d) respectively in the previous $k-1$ trajectories. UCT selects the action to simulate in order to extend the trajectory greedily with respect to this summed score. Once the input-parameter number of trajectories are generated each to maximum depth, UCT returns the exploitation term for each action at the root node (which is the current state it is planning an action for) as its estimate of the utility of taking that action in the current state of the game. UCT has the nice theoretical property that the number of simulation steps (number of trajectories × maximum-depth) needed to ensure any bound on the loss of following the UCT-based policy is independent of the size of the state space. This result captures the fact that the use of UCT avoids the perception problem, but at the cost of requiring substantial computation for every time step of action selection because it never builds an explicit policy.

Performance Gap & Our Opportunity. The opportunity for this chapter arises from the following observations. The model-free RL agents for Atari games are fast (indeed faster than real-time, e.g., the CNN-based approach from this chapter takes 10^{-4} seconds to select an action on our computer) while the UCT-based planning agents are several orders of magnitude slower (much slower than real-time, e.g., they take seconds to select an action on the same computer). On the other hand, the performance of UCT-based planning agents is much better than the performance of model-free RL agents (this will be evident in our results below). Our goal is to develop methods that retain the DL advantage of not needing hand-crafted features and the online real-time play ability of the model-free RL agents by exploiting data generated by UCT-planning agents.

2.2 Methods for Combining UCT-based RL and DL

We first describe the baseline UCT agent, and then three agents that instantiate different methods of combining the UCT agent with DL. Recall that in keeping with the goal of building general-purpose methods as in the DQN work we impose the constraint of reusing the same input representations, the same function approximation architecture, and the same planning method for all the games.

2.2.1 Baseline UCT Agent that Provides Training Data

This agent requires no training. It does, however, require specification of its two parameters, the number of trajectories and the maximum-depth. Recall that our proposed new agents will all use data from this UCT-agent to train a CNN-based policy and so it is reasonable that the resulting performance of our proposed agents will be worse than that of the UCT-agent. Therefore, in our experiments we set these two parameters large enough to ensure that they outscore the published DQN scores, but not so large that they make our computational experiments unreasonably slow. Specifically, we elected to use 300 as maximum-depth and 10000 as number of trajectories for all games but two. Pong turns out to be a much simpler game and we could reduce the number of trajectories to 500, and Enduro turned out to have more distal rewards than the other games and so we used a maximum-depth of 400. As will be evident from the results in Section 2.3, this allowed the UCT agent to significantly outperform DQN in all games but Pong in which DQN already performs almost perfectly. We emphasize that the UCT agent does not meet our goal of real-time play. For example, to play a game just 800 times with the UCT agent (we do this to collect training data for our agent's below) takes a few days on a recent multicore computer for each game.

2.2.2 Our Three Methods and their Corresponding Agents

Method 1: UCT to Regression (for **U**CT to **C**NN via **R**egression). The key idea is to use the action values computed by the UCT-agent to train a regression-based CNN. The following is done for each game. Collect 800 UCT-agent runs by playing the game 800 times from start to finish using the UCT agent above. Build a dataset (table) from these runs as follows. Map the last four frames of each state along each trajectory into the action-values of all the actions as computed by UCT. In some states of Atari games, the controller actions do not have any effects. For example, when the player's avatar is falling from the cubes in Q*Bert, no actions can change the next state of the game. Our UCT implementation would assign equal action values to all different actions whenever this kind of states occur. Thus, we filter out all this kind of states in the data collection for our methods. The detection of this kind of states is trivial in UCT because UCT would try all different actions of its current root node and compare the next states. If the next states of different actions are the same, such root node is excluded from the data collection and no look-ahead tree building is continued. A random action would be taken in those states in real execution. The collected training data is used to train the CNN via regression (see below for CNN details). The UCT to Regression-agent uses the CNN learned by this training procedure to select actions during evaluation.

Method 2: UCT to Classification (for **U**CT to **C**NN via **C**lassification). The key idea is to use the action choice computed by the UCT-agent (selected greedily from action-values) to train a classifier-based CNN. The following is done for each game. Collect 800 UCT-agent runs as above. These runs yield a table in which the rows correspond to the last four frames at each state along each trajectory and the single column is the choice of action that is best according to the UCT-agent at that state of the trajectory. This training data is used to train the CNN via multiclass classification (see below for CNN details). The UCT to Classification-agent uses the

CNN-classifier learned by this training procedure to select actions during evaluation.

One potential issue with the above two agents is that the training data's input distribution is generated by the UCT-agent while during testing the UCT to Regression and UCT to Classification agents will perform differently from the UCT-agent and thus could experience an input distribution quite different from that of the UCT-agent's. This could limit the testing performance of the UCT to Regression and UCT to Classification agents. Thus, it might be desirable to somehow bias the distribution over inputs to those likely to be encountered by these agents; this observation motivates our next method.

Method 3: UCT to Classification Interleaved (UCC-I, for **U**CT to **C**NN via **C**lassification - **I**nterleaved). The key idea is to focus UCT planning on that part of the state space experienced by the (partially trained) CNN player. The method accomplishes this by interleaving training and data collection as follows.[2] Collect 200 UCT-agent runs as above; these will obviously have the same input distribution concern raised above. The data from these runs is used to train the CNN via classification just as in the UCT to Classification-agent's method (we do not do this for the UCT to Regression-agent because as we show below it performs worse than the UCT to Classification-agent). The trained CNN is then used to decide action choices in collecting further 200 runs (though 5% of the time a random action is chosen to ensure some exploration). At each state of the game along each trajectory, UCT is asked to compute its choice of action and the original data set is augmented with the last four frames for each state as the rows and the column as UCT's action choice. This 400 trajectory dataset's input distribution is now potentially different from that of the UCT-agent. This dataset is used to train the CNN again via classification. This

[2] Our UCT to Classification Interleaved method is a special case of DAgger (*Ross et al.* (2011)) (in the use of a CNN-classifier and in the use of specific choices of parameters $\beta_1 = 1$, and for $i > 1$, $\beta_i = 0$). As a small point of difference, we note that our emphasis in this chapter was in the use of CNNs to avoid the use of hand-crafted domain specific features, while the empirical work for DAgger did not have the same emphasis and so used hand-crafted features.

interleaved procedure is repeated until there are a total of 800 runs worth of data in the dataset for the final round of training of the CNN. The UCT to Classification Interleaved agent uses the final CNN-classifier learned by this training procedure to select actions during testing.

In order to focus our empirical evaluation on the contribution of the non-DL part of our three new agents, we reused the same convolutional neural network architecture as used in the DQN work (we describe this architecture in brief detail below). The DQN work modified the reward functions for some of the games (by saturating them at $+1$ and -1) while we use unmodified reward functions (these only play a role in the UCT-agent components of our methods and not in the CNN component). We also follow DQN's frame-skipping techniques: the agent sees and selects actions on every k^{th} frame instead of every frame ($k = 3$ for Space Invaders and $k = 4$ for all other games), and the latest chosen-action is repeated on subsequently-skipped frames.

2.2.3 Details of Data Preprocessing and CNN Architecture

Preprocessing (identical to DQN to the best of our understanding). Raw Atari game frames are 160×210 pixel images with a 128-color palette. We convert the RGB representation to gray-scale and crop a 160×160 region of the image that captures the playing area, and then the cropped image is down-sampled to 84×84 in order to reuse DQN's CNN architecture. This procedure is applied to the last 4 frames associated with a state and stacked to produce a $84 \times 84 \times 4$ preprocessed input representation for each state. We subtracted the pixel-level means and scale the inputs to lie in the range [-1, 1]. We shuffle the training data to break the strong correlations between consecutive samples, which therefore reduces the variance of the updates.

CNN Architecture. We adapted the deep neural network architecture from DQN (*Mnih et al.* (2013)) for our agents, except for the nonlinearity units of convolu-

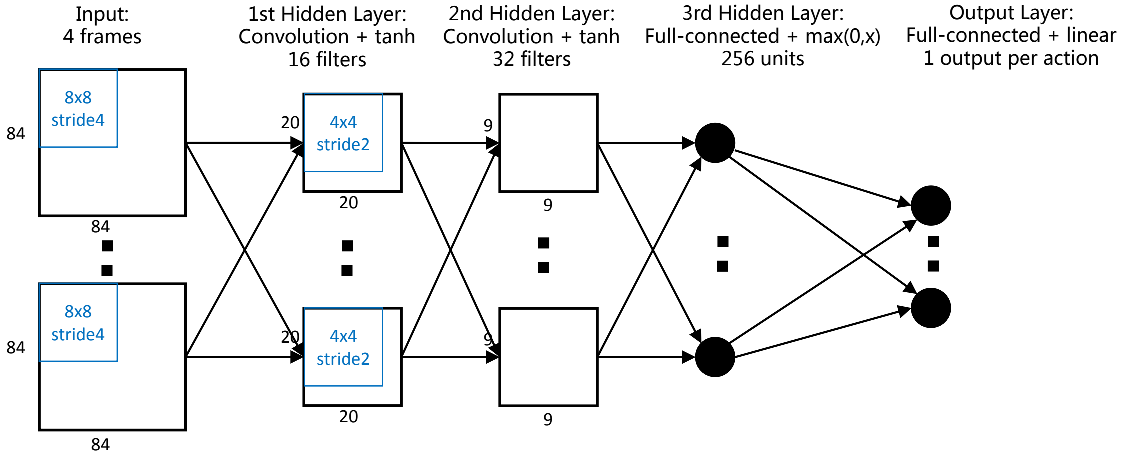


Figure 2.3: The CNN architecture from DQN (*Mnih et al.* (2013)) that we adopt in our agents. See text for details.

tional layers.[3] As depicted in Figure 2.3, our network consists of three hidden layers. The input to the neural network is an $84 \times 84 \times 4$ image produced by the preprocessing procedure above. The first hidden layer convolves 16, 8×8, filters with stride 4 with the input image and applies a nonlinearity (tanh). The second hidden layer convolves 32, 4×4, filters with stride 2 again followed by a nonlinearity (tanh). The final hidden layer is fully connected and consists of 256 rectified linear units (relu). In the multi-regression-based agent (UCT to Regression), the output layer is a fully connected linear layer with a single output for each valid action. In the classification-based agents (UCT to Classification, UCT to Classification Interleaved), a softmax (instead of linear) function is applied to the final output layer. We refer the reader to the DQN paper for further detail.

[3] We also tested with rectified linear nonlinearity for convolutional layers, but did not find significant differences in the game performance.

Table 2.1: Performance (game scores) of the four real-time game playing agents, where UCR is short for UCT to Regression, UCC is short for UCT to Classification, and UCC-I is short for UCT to Classification Interleaved.

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175 (5.63)	558 (14)	19 (0.3)	11574(44)	2273 (23)	672 (5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388 (4.6)	215 (6.69)	601 (11)	19 (0.14)	13189 (35.3)	2701 (6.09)	670 (4.24)
-best	10732	413	1026	21	29900	6100	910
-greedy	5702	380	741	21	20025	2995	692
UCR	2405 (12)	143 (6.7)	566 (10.2)	19 (0.3)	12755 (40.7)	1024 (13.8)	441 (8.1)

Table 2.2: Performance (game scores) of the off-line UCT game playing agent.

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
UCT	7233	406	788	21	18850	3257	2354

2.3 Experimental Results

First we present our main performance results and then present some visualizations to help understand the performance of our agents.[4] In Table 2.1 we compare and contrast the performance of the four real-time game playing agents, three of which (UCT to Regression, UCT to Classification, and UCT to Classification Interleaved) we implemented and evaluated; the performance of the DQN was obtained from *Mnih et al.* (2013).

The columns correspond to the seven games named in the header, and the rows correspond to different assessments of the four agents. Throughout the table, the numbers in parentheses are standard-errors. The DQN row reports the average performance (game score) of the DQN agent (a random action is chosen 5% of the time during testing). The DQN-*best* row reports the best performance of the DQN agent

[4] Our code and game play videos are available at: sites.google.com/site/nips2014atari/.

over all the attempts at each game. Comparing the performance of the UCT to Classification and UCT to Regression agents (both use 5% exploration), we see that the UCT to Classification agent either competes well with or significantly outperforms the UCT to Regression agent. The result is mainly caused by the fact that the action values of non-greedy action in UCT's estimation are much noisier than the empirical greedy one. UCT allocates most of the sampling trajectories on empirical greedy actions and non-greedy actions may only be sampled a tiny fraction of the total trajectories. UCT to Regression approach considers the estimation of all actions, so it is more influenced by the noise. Another factor in explaining the difference between UCT to Regression and UCT to Classification is that policies are generally easier to generalize than state/action values. More importantly the UCT to Classification agent outperforms the DQN agent in all games but Pong (in which both agents do nearly perfectly because the maximum score in this game is 21). The percentage-performance gain of UCT to Classification over DQN is quite large for most games. Similar gains are obtained in the comparison of UCT to Classification-*best* to DQN-*best*.

We used 5% exploration in our agents to match what the DQN agent does. In any case, the effect of this randomness in action-selection will differ across games (based, e.g., on whether a wrong action can be terminal). Thus, we also present results for the UCT to Classification-*greedy* agent in which we don't do any exploration. As seen by comparing the rows corresponding to UCT to Classification and UCT to Classification-*greedy*, the latter agent always outperforms the former and in four games (Breakout, Enduro, Q*Bert, and Seaquest) achieves further large-percentage improvements.

Table 2.2 gives the performance of our non-realtime UCT agent (again, with 5% exploration). As discussed above we selected UCT-agent's parameters to ensure that this agent outperforms the DQN agent allowing room for our agents to perform in

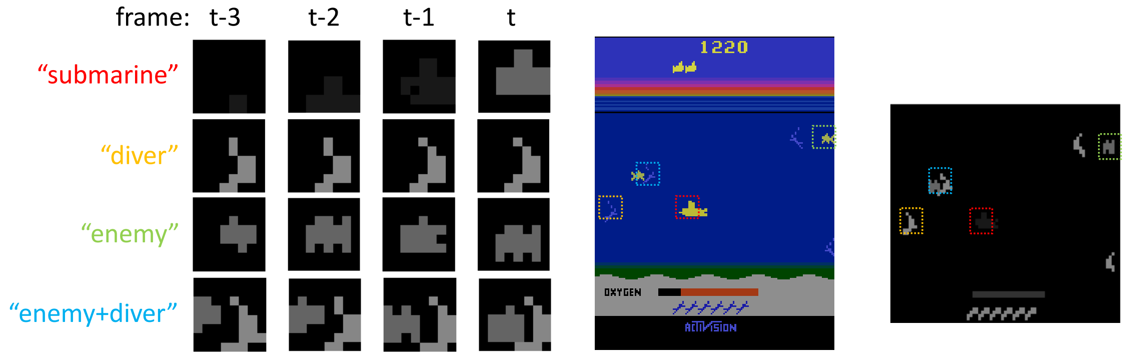


Figure 2.4: Visualization of the first-layer features learned from Seaquest. (Left) visualization of four first-layer filters; each filter covers four frames, showing the spatio-temporal template. (Middle) a captured screen. (Right) gray-scale version of the input screen which is fed into the CNN. Four filters were color-coded and visualized as dotted bounding boxes at the locations where they get activated. This figure is best viewed in color.

the middle.

Finally, recall that the UCT to Classification Interleaved agent was designed so that its input distribution during training is more likely to match its input distribution during evaluation and we hypothesized that this would improve performance relative to UCT to Classification. Indeed, in all games but B. Rider, Pong and S.Invaders in which the two agents perform similarly, UCT to Classification Interleaved significantly outperforms UCT to Classification. The same holds when comparing UCT to Classification Interleaved-best and UCT to Classification-best as well as UCT to Classification Interleaved-greedy and UCT to Classification-greedy.

Overall, the average game performance of our best performing agent (UCT to Classification Interleaved) is significant higher than that of DQN for most games, such as B.Rider (32%), Breakout (28%), Enduro (28%), Q*Bert (580%), Sequest (58%) and S.Invaders (15%).

In a further preliminary exploration of the effectiveness of the UCT to Classification Interleaved in exploiting additional computational resources for generating

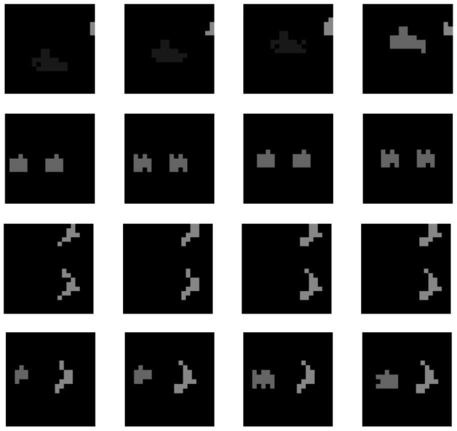
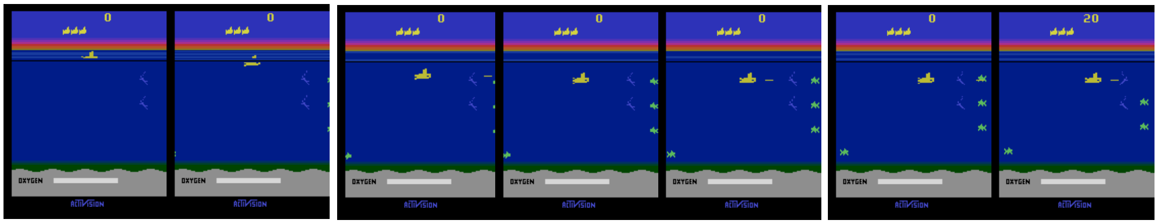


Figure 2.5: Visualization of the second-layer features learned from Seaquest.

UCT runs, on the game Enduro we compared UCT to Classification and UCT to Classification Interleaved where we allowed each of them twice the number of UCT runs used in producing the Table 2.1 above, i.e., 1600 runs while keeping a batch size of 200. The performance of UCT to Classification improves from 558 to 581 while the performance of UCT to Classification Interleaved improves from 601 to 670, i.e., the interleaved method improved more in absolute and percentage terms as we increased the amount of training data. This is encouraging and is further confirmation of the hypothesis that motivated the interleaved method, because the interleaved input distribution would be even more like that of the final agent with the larger data set.

Learned Features from Convolutional Layers. We provide visualizations of the learned filters in order to gain insight on what the CNN learns. Specifically, we apply the "optimal stimuli" method (*Erhan et al.* (2009); *Le et al.* (2012)) to visualize the features CNN learned after training. The method picks the input image patches that generate the greatest responses after convolution with the trained filters. We select 8 × 8 × 4 input patches to visualize the first convolutional layer features and 20 × 20 × 4 to visualize the second convolutional layer filters. Note that these



Step 69: FIRE Step 70: DOWN+FIRE Step 74:DOWN+FIRE Step 75:RIGHT+FIRE Step 76:RIGHT+FIRE Step 78: RIGHT+FIRE Step 79:DOWN+FIRE

Figure 2.6: A visualization of the UCT to Classification agent's policy as it kills an enemy agent.

patch sizes correspond to receptive field sizes of the learned features in each layer. In Figure 2.4, we show four first-layer filters of the CNN trained from Seaquest for the UCT to Classification-agent. Specifically, each filter covers four frames of 8×8 pixels, which can be viewed as a spatio-temporal template that captures specific patterns and their temporal changes. We also show an example screen capture and visualize where the filters get activated in the gray-scale version of the image (which is the actual input to the CNN model). The visualization suggests that the first-layer filters capture "object-part" patterns and their temporal movements.

Figure 2.5 visualizes the four second-layer features via the optimal stimulus method, where each row corresponds to a filter. We can see that the second-layer features capture bigger spatial patterns (often covering beyond the size of individual objects), while encoding interactions between objects, such as two enemies moving together, and a submarine moving along a direction.

Overall, these qualitative results suggest that the CNN learns relevant patterns useful for game playing.

Visualization of Learned Policy. Here we present visualizations of the policy learned by the UCT to Classification agent with the aim of illustrating both what it does well and what it does not.

Figure 2.6 shows the policy learned by UCT to Classification to destroy nearby enemies. The CNN changes the action from "Fire" to "Down+Fire" at time step 70 when the enemies first show up at the right columns of the screen, which will move the submarine to the same horizontal position of the closest enemy. At time step 75, the submarine is at the horizontal position of the closest enemy and the action changes to “Right+Fire”. The “Right+Fire” action is repeated until the enemy is destroyed at time step 79. At time step 79, the predicted action is changed to “Down+Fire” again to move the submarine to the horizontal position of the next closest enemy. This shows the UCT to Classification agent’s ability to deal with delayed reward as it learns to take a sequence of unrewarded actions before it obtains any reward when it finally destroys an enemy.

Figure 2.6 also shows a shortcoming in the UCT to Classification agent’s policy, namely it does not purposefully take actions to save a diver (although saving *many* divers can lead to a large reward). For example, at time step 69, even though there are two divers below and to the right of the submarine (our agent), the learned policy does not move the submarine downward. This phenomenon was observed frequently. The reason for this shortcoming is that it can take a large number of time steps to capture 6 divers and bring them to surface (bringing fewer divers to the surface does not yield a reward); this takes longer than the planning depth of UCT, and UCT fails to do it for most of the times. Thus, even our UCT agent does not purposefully save divers and thus the training data collected reflects that defect which is then also present in the play of the UCT to Classification (and UCT to Classification Interleaved) agent.

2.4 Conclusion

UCT-based planning agents are unrealistic for Atari game play in at least two ways. First, to play the game they require access to the state of the game which is unavailable to human players, and second they are orders of magnitude slower than realtime. On the other hand, by slowing the game down enough to allow UCT to play leads to the highest scores on the games they have been tried on. Indeed, by allowing UCT more and more time (and thus allowing for larger number of trajectories and larger maximum-depth) between moves one can presumably raise the score higher and higher. We identified a gap between the UCT-based planning agent's performance and the best realtime player DQN's performance and developed new agents to partially fill this gap. Our main applied result is that at the time of the writing of this chapter we have the best realtime Atari game playing agents on the same 7 games that were used to evaluate DQN. Indeed, in most of the 7 games our best agent beats DQN significantly. Another result is that at least in our experiments training the CNN to learn a classifier that maps game observations to actions was better than training the CNN to learn a regression function that maps game observations to action-values (we intend to do further work to confirm how general this result is on ALE). Finally, we hypothesized that the difference in input distribution between the UCT agent that generates the training data and the input distribution experienced by our learned agents would diminish performance. The UCT to Classification Interleaved agent we developed to deal with this issue indeed performed better than the UCT to Classification agent indirectly confirming our hypothesis and solving the underlying issue.

CHAPTER III

Deep Learning for Reward Design to Improve Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) methods have proven powerful in planning for large scale sequential decision-making problems such as classic board games and video games. For example, the strongest Reinforcement Learning programs for most classic games are based on MCTS methods, such as AlphaGo for Go(*Silver et al.* (2016)), Meep for Chess(*Veness et al.* (2009)), Logistello for Othello(*Buro* (2002)), Maven for Scrabble(*Sheppard* (2002)), and Chinook for Checkers(*Schaeffer et al.* (1992)). To address the delayed reward issue (the outcome of action choice in terms of rewards is delayed in time), MCTS methods build look-ahead trees to estimate the action values. The look-ahead tree building procedure is usually time-consuming and memory demanding. More importantly, MCTS methods suffer when the planning depth and sampling trajectories are limited compared to the delay of rewards or when the rewards are sparse.

This chapter presents an end-to-end reward design approach to address such computational constraints of MCTS methods. The reward design work builds on PGRD (policy-gradient for reward-design; *Sorg et al.* (2010a)), a method for learning reward-bonus functions for use in planning. Previous applications of PGRD have been limited in a few ways: 1) they have been applied to small RL problems; 2) they have required careful hand-construction of features that define a reward-bonus function space to be searched over by PGRD; 3) they have limited the reward-bonus function space to be a linear function of hand-constructed features; and 4) they have high-variance in the gradient estimates (this issue was not apparent in earlier work because of the small size of the domains). In this chapter, we address all of these limitations by developing PGRD-DL, a PGRD variant that automatically learns features over raw perception inputs via a multi-layer convolutional neural network (CNN), and uses a variance-reducing form of gradient. We show that PGRD-DL can improve performance of UCT on multiple ATARI games.

In terms of deep reinforcement learning algorithms, this chapter offers a novel means of combining Deep Learning (DL; see *Bengio* (2009); *Schmidhuber* (2015) for surveys) and Reinforcement Learning (RL), with an application to ATARI games. There has been a flurry of recent work on combining DL and RL on ATARI games, including the seminal work using DL as a function approximator for Q-learning (*Mnih et al.* (2015); *Hausknecht and Stone* (2015)), the use of UCT-based planning to provide policy-training data for a DL function approximator (*Guo et al.* (2014)), the use of DL to learn transition-models of ATARI games to improve exploration in Q-learning (*Oh et al.* (2015); *Stadie et al.* (2015)), and the use of DL as a parametric representation of policies to improve via policy-gradient approaches (*Schulman et al.* (2015)).

In contrast, the work presented here uses DL as a function approximator to learn reward-bonus functions from experience to mitigate computational limitations in UCT (*Kocsis and Szepesvári* (2006)), a Monte Carlo Tree Search (MCTS) algorithm. In large-scale sequential decision-making problems, UCT often suffers because of limitations on the number of trajectories and planning depth required to keep the method computationally tractable (*Srinivasan et al.* (2015)). The key contribution of this chapter is a new method for improving the performance of UCT planning in such challenging settings, exploiting the powerful feature-learning capabilities of DL.

3.1 Related Work

ALE as a challenging testbed for RL. The Arcade Learning Environment (ALE) includes an ATARI 2600 emulator and about 50 games (*Bellemare et al.* (2013b)). These games represent a challenging combination of perception and policy selection. All of the games have the same high-dimensional observation screen, a 2D array of 7-bit pixels, 160 pixels wide by 210 pixels high. The action space depends on the game but consists of at most 18 discrete actions. Designing agents to learn to play ATARI games is challenging due to the high-dimensionality and partial observability of the perceptions, as well as the sparse and often highly-delayed nature of the rewards. There are a number of approaches to building ATARI game players, including some that do not use DL (e.g., *Bellemare et al.* (2012a), *Bellemare et al.* (2012b), *Bellemare et al.* (2013a)). Among these are players that use UCT to plan with the ALE emulator as a model (*Bellemare et al.* (2013b)), an approach that we build on here.

Combining DL and RL. Deep Learning has emerged as a powerful technique for learning feature representations from raw input data in a compositional hierarchy, where low-level features are learned to encode low-level statistical dependencies (e.g. edges in images), and higher-level features encode higher-order dependencies of the lower-level features (e.g. object parts; *Lee et al.* (2009)). Combining DL and RL methods together thus offers an opportunity to address the high-dimensional perception, partial observability and sparse and delayed reward challenges in designing ATARI game playing agents. Indeed, as summarized above, recent work has started to do just that, though research into combining neural networks and RL is older than the recent interest in ALE (e.g., *Koutník et al.* (2013); *Tesauro* (1994)).

UCT. UCT is widely used to do planning in large scale sequential decision problems because the complexity of the action selection step is independent of the size of the state space (allowing it to be applied to ATARI games, for example).

UCT has three parameters: the number of trajectories, the maximum-depth, and an exploration parameter. In general, the larger the trajectory and depth parameters are, the better the performance is, though at the expense of increased computation at each action selection. UCT computes a score for each possible action a in state-depth pair (s, d) as the sum of two terms: an exploitation term that is the Monte Carlo average of the discounted sum of rewards obtained from experiences with state-depth pair (s, d) in the previous k-1 trajectories, and an exploration term that encourages sampling infrequently taken actions.[1] UCT selects the action to simulate in order to extend the trajectory greedily with respect to the summed score. Once the number of sampled trajectories reaches the parameter value, UCT returns the exploitation term for each action at the root node as its estimate of the utility of taking that action in the current state of the game.

Although they have been successful in many applications, UCT and more generally MCTS methods also have limitations: they suffer when the number of trajectories and planning depth are limited relative to the size of the domain, or when the rewards are sparse, which creates a kind of needle-in-a-haystack problem for search.

Mitigating UCT limitations. Agent designers often build in heuristics for overcoming these UCT drawbacks. As one example, in the case of limited planning depth, the classical Leaf-Evaluation Heuristic (LEH) (*Shannon* (1950)) adds a heuristic value to the estimated return at the leaf states. However, the value to compensate for the missing subtree below the leaf is often not known in practice. Many other approaches improve UCT by mechanisms for generalization of returns and visit statistics across

[1] Specifically, the exploration term is $c\sqrt{\log n(s,d)/n(s,a,d)}$ where $n(s,d)$ and $n(s,a,d)$ are the number of visits to state-depth pair (s,d), and of action a in state-depth pair (s,d) respectively in the previous $k-1$ trajectories, and c is the exploration parameter.

states: e.g., the Transposition Tables of *Childs et al.* (2008), the Rapid Action Value Estimation of *Gelly and Silver* (2011), the local homomorphisms of *Jiang et al.* (2014), the state abstractions of *Hostetler et al.* (2014), the local manifolds of *Srinivasan et al.* (2015) and the most famous policy and value networks in AlphaGo of *Silver et al.* (2016).

In this chapter, we explore learning a reward-bonus function to mitigate the computational bounds on UCT. Through deep-learned features of state, our method also provides a generalization mechanism that allows UCT to exploit useful knowledge gained in prior planning episodes.

Reward design, optimal rewards, and PGRD *Singh et al.* (2010) proposed a framework of optimal rewards which allows the use of a reward function *internal* to the agent that is potentially different from the *objective* (or task-specifying) reward function. They showed that good choices of internal reward functions can mitigate agent limitations.[2] *Sorg et al.* (2010b) proposed PGRD as a specific algorithm that exploits the insight that for planning agents a reward function implicitly specifies a policy, and adapts policy gradient methods to search for good rewards for UCT-based agents when its planning depth or number of trajectories is limited.

Unlike previous applications of PGRD in which the space of reward-bonus functions was limited to linear functions of hand-coded state-action-features, we use PGRD with a multi-layer convolutional neural network to automatically learn features from raw perception as well as to adapt the non-linear reward-bonus function parameters. We also adopt a variance-reducing gradient method to improve PGRD's performance. The new method improves UCT's performance on multiple ATARI

[2] Others have developed methods for the design of rewards under different settings. Potential-based Reward Shaping (*Ng et al.* (1999); *Asmuth et al.* (2008)) offers a space of reward functions with the property that an optimal policy for the original reward function remains an optimal policy for each reward function in the space. This allows the designer to heuristically pick a reward function for other properties, such as impact on learning speed. In some cases, reward functions are simply unknown, in which case inverse-RL (*Ng and Russell* (2000)) has been used to infer reward functions from expert behavior. Yet others have explored the use of queries to a human expert to elicit preferences (*Chajewska et al.* (2000)).

games compared to UCT without the reward bonus. Combining PGRD and Deep Learning in this way should make adapting rewards for MCTS algorithms far more widely and practically applicable than before.

3.2 PGRD-DL: Policy-Gradient for Reward Design with Deep Learning

Understanding PGRD-DL requires understanding three major components. First, PGRD-DL uses UCT differently than usual in that it employs an internal reward function that is the sum of a reward-bonus and the usual objective reward (in ATARI games, the change in score). Second, there is a CNN-based parameterization of the reward-bonus function. Finally, there is a gradient procedure to train the CNN-parameters. We describe each in turn.

UCT with internal rewards. As described in the Related Work section above, in extending a trajectory during planning, UCT computes a score that combines a UCB-based *exploration term* that encourages sampling infrequently sampled actions with an *exploitation term* computed from the trajectories sampled thus far. A full H-length trajectory is a sequence of state-action pairs: $s_0 a_0 s_1 a_1 ... s_{H-1} a_{H-1}$. UCT estimates the exploitation-term value of a state, action, depth tuple (s, a, d) as the average return obtained after experiencing the tuple (non-recursive form):

$$Q(s,a,d) = \sum_{i=1}^{N} \frac{I_i(s,a,d)}{n(s,a,d)} \sum_{h=d}^{H-1} \gamma^{h-d} R(s_h^i, a_h^i) \tag{3.1}$$

where N is the number of trajectories sampled, γ is the discount factor, $n(s,a,d)$ is the number of times tuple (s,a,d) has been sampled, $I_i(s,a,d)$ is 1 if (s,a,d) is in the i^{th} trajectory and 0 otherwise, s_h^i is the h^{th} state in the i^{th} trajectory and a_h^i is the h^{th} action in the i^{th} trajectory.

The difference between the standard use of UCT and its use in PGRD-DL is in the choice of the reward function in Equation 3.1. To emphasize this difference we use the notation R^O to denote the usual *objective* reward function, and R^I to denote the new *internal* reward function. Specifically, we let

$$R^I(s, a; \theta) = \mathrm{CNN}(s, a; \theta) + R^O(s, a) \tag{3.2}$$

where the reward-bonus that is added to the objective reward in computing the internal reward is represented via a multi-layered convolution neural network, or CNN, mapping from state-action pairs to a scalar; θ denotes the CNN's parameters, and thus the reward-bonus parameters. To denote the use of internal rewards in Equation 3.1 and to emphasize its dependence on the parameters θ, we will hereafter denote the Q-value function as $Q^I(\cdot, \cdot, \cdot; \theta)$. Note that the reward bonus in Equation 3.2 is distinct from (and does not replace) the exploration bonus used by UCT during planning.

CNN parameterization of reward-bonuses. PGRD-DL is capable of using any kind of feed-forward neural network to represent the reward bonus functions. The specifics of this are defined in the Experiment Setup section below.

Gradient procedure to update reward-bonus parameters. When UCT finishes generating the specified number of trajectories (when planning is complete), the greedy action is

$$a = \arg\max_b Q^I(s, b, 0; \theta) \tag{3.3}$$

where the action values of the current state are $Q^I(s, ., 0; \theta)$. To allow for gradient calculations during training of the reward-bonus parameters, the UCT agent executes actions according to a softmax distribution given the estimated action values (the temperature parameter is omitted):

$$\mu(a|s;\theta) = \frac{\exp Q^I(s,a,0;\theta)}{\sum_b \exp Q^I(s,b,0;\theta)}, \tag{3.4}$$

where μ denotes the UCT agent's policy.

Even though internal rewards are used to determine the UCT policy, only the task-specifying objective reward R^O is used to determine how well the internal reward function is doing. In other words, the performance of UCT with the internal reward function is measured over the experience sequence that consists of the actual executed actions and visited states: $h_T = s_0a_0s_1a_1...s_{T-1}a_{T-1}$ in terms of objective-return $u(.)$:

$$u(h_T) = \sum_{t=0}^{T-1} R^O(s_t, a_t) \tag{3.5}$$

where s_t and a_t denote the actual state and action at time t. Here we assume that all tasks are episodic, and the maximum length of an experience sequence is T. The expected objective-return is a function of the reward bonus function parameters θ because the policy μ depends on θ, and in turn the policy determines the distribution over state-action sequences experienced.

The PGRD-DL objective in optimizing reward bonus parameters is maximizing the expected objective-return of UCT:

$$\theta^* = \arg\max_\theta U(\theta) = \arg\max_\theta \mathbb{E}\{u(h_T)|\theta\}. \tag{3.6}$$

The central insight of PGRD was to consider the reward-bonus parameters as policy parameters and apply stochastic gradient ascent to maximize the expected return. Previous applications of PGRD used linear functions of hand-coded state-action-features as reward-bonus functions. In this chapter, we first applied PGRD with a multi-layer convolutional neural network to automatically learn features from raw perception as well as to adapt the non-linear reward-bonus parameters. However, empirical results showed that the original PGRD could cause the CNN parameters to diverge and cause performance degeneration due to large variance in the policy gradient estimation. We therefore adapted a variance-reduction policy gradient method GARB (GPOMDP with Average Reward Baseline; *Weaver and Tao* (2001)) to solve this drawback of the original PGRD. GARB optimizes the reward-bonus parameters to maximize the expected objective-return as follows:

$$\begin{aligned}\nabla_\theta U(\theta) &= \nabla_\theta \, \mathbb{E}\{u(h_T)|\theta\} && (3.7)\\ &= \mathbb{E}\{u(h_T)\sum_{t=0}^{T-1}\frac{\nabla_\theta \mu(a_t|s_t;\theta)}{\mu(a_t|s_t;\theta)}\} && (3.8)\end{aligned}$$

Thus $u(h_T)\sum_{t=0}^{T-1}\frac{\nabla_\theta \mu(a_t|s_t;\theta)}{\mu(a_t|s_t;\theta)}$ is an unbiased estimator of the objective-return gradient. GARB computes an eligibility trace vector $\mathbf{e}$ to keep track of the gradient vector $\mathbf{g}$ at time t:

$$\begin{aligned}\mathbf{e_{t+1}} &= \beta\mathbf{e_t} + \frac{\nabla_\theta \mu(a_t|s_t;\theta)}{\mu(a_t|s_t;\theta)} && (3.9)\\ \mathbf{g}_{t+1} &= \mathbf{g}_t + (r_t - b)\mathbf{e_{t+1}} && (3.10)\end{aligned}$$

where $\beta \in [0, 1)$ is a parameter controlling the bias-variance trade-off of the gradient estimation, $r_t = R^O(s_t, a_t)$ is the immediate objective-reward at time t, b is a reward baseline and it equals the running average of r_t, and $\mathbf{g_T}$ is the gradient estimate vector. We use the gradient vector to update the reward parameters θ when a terminal state is reached; at the end of the j^{th} episode, $\mathbf{g_T}$ is used to update θ using an existing stochastic gradient based method, ADAM (*Kingma and Ba* (2015)), as described below.

Since the reward-bonus function is represented as a feed-forward network, backpropagation can compute the gradient of the reward-bonus function parameters efficiently, i.e. $\frac{\nabla_\theta \mu(a_t|s_t;\theta)}{\mu(a_t|s_t;\theta)}$. In order to apply BackProp, we need to compute the derivative of policy μ with respect to the reward $R^I(s_h^i, a_h^i; \theta)$ in the i^{th} sampling trajectory at depth h in UCT planning:

$$\delta_t^{r(i,h)} = \frac{1}{\mu(a_t|s_t)} \frac{\partial \mu(a_t|s_t)}{\partial R^I(s_h^i, a_h^i)} \tag{3.11}$$

$$= \frac{1}{\mu(a_t|s_t)} \sum_b \frac{\partial \mu(a_t|s_t)}{\partial Q^I(s_t, b, 0)} \frac{\partial Q^I(s_t, b, 0)}{\partial R^I(s_h^i, a_h^i)} \tag{3.12}$$

$$= \sum_b (I(a_t = b) - \mu(b|s_t)) \frac{I_i(s_t, b, 0)}{n(s_t, b, 0)} \gamma^h \tag{3.13}$$

where $I(a_t = b) = 1$ if a_t equals b and 0 otherwise. Thus the derivative of any parameter θ_k in the reward parameters can be represented as:

$$\delta_t^{\theta_k} = \sum_{i,h} \delta_t^{r(i,h)} \frac{\partial R^I(s_h^i, a_h^i)}{\partial \theta_k} \tag{3.14}$$

where $\frac{\partial R^I(s_h^i, a_h^i)}{\partial \theta_k}$ is determined by (s_h^i, a_h^i) and the CNN, and can be computed efficiently using standard BackProp.

What does PGRD-DL learn? We emphasize that the only thing that is learned from experience during repeated application of PGRD-DL planning procedure is the reward-bonus function. All the other aspects of the PGRD-DL procedure remain fixed throughout learning.

3.3 Experiment Setup: Evaluating PGRD-DL on ATARI Games

We evaluated PGRD-DL on 25 ATARI games (Table 3.1). All the ATARI games involve moving a game agent in a 2-D space, but otherwise have very different dynamics.

UCT objective reward and planning parameters. As is standard in RL work on ATARI games, we take the objective reward for each state to be the difference in the game score between that state and the previous state. We rescale the objective reward to ease the training of PGRD. We assign +1 for positive rewards, and -1 for negative rewards. A game-over state or life-losing state is considered a terminal state in UCT planning and PGRD training. Evaluation trajectories only consider game-over states as terminal states.

All UCT baseline agents in our experiments sample 100 trajectories of depth 100 frames. Normally UCT does planning for every visited state. However, for some states in ATARI games the next state and reward is the same no matter which action is chosen (for example, when the Q*Bert agent is falling) and so UCT planning is a waste of computation. In such states our agents do not plan but instead choose a random action. Those states are also skipped in the policy gradient computation in learning reward bonuses. The UCB-parameter is 0.1 and the discount factor $\gamma = 0.99$. Following *Mnih et al.* (2015), we use a simple frame-skipping technique to save computations: the agent selects actions on every 4^{th} frame instead of every frame, and the last action is repeated on skipped frames. We did not apply PGRD to ATARI games in which UCT achieves the highest possible score, such as Pong and Boxing.

Screen image preprocessing. The last four game screen images are used as input for the CNN. The 4 frames are stacked in channels. The game screen images (210 × 160) are downsampled to 84 × 84 pixels and gray-scaled. Each image is further preprocessed by pixel-wise mean removal. The pixel-wise mean is calculated over ten game trajectories of a uniformly-random policy.

Convolutional network architecture. The same network architecture is used for all games. The network consists of 3 hidden layers. The first layer convolves 16, 8×8 filters with stride 4. The second hidden layer convolves 32, 4×4 filters with stride 2.

The third hidden layer is a full-connected layer with 256 units. Each hidden layer is followed by a rectifier nonlinearity (ReLU). The output layer has one unit per action.

PGRD-DL learning parameters. After computing the gradients of CNN parameters, we use ADAM to optimize the parameters of the CNN. ADAM is an adaptive stochastic gradient optimization method to train deep neural networks (*Kingma and Ba* (2015)). We use the default hyper-parameters of ADAM[3]. We set $\beta = 0.99$ in GARB. Thus the only remaining hyper-parameter is the learning rate for ADAM, which we selected from a candidate set $\{10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$ by identifying the rate that produced the greatest sum of performance improvements after training 1000 games on Ms. Pacman and Q*Bert. The selected learning rate of 10^{-4} served as the initial learning rate for PGRD-DL. The learning rate was then lowered during learning by dividing it by 2 every 1000 games.

3.4 Experiment Results

We allowed PGRD-DL to adapt the bonus reward function for at most 5000 games or at most 10 million steps, stopping the training whenever one condition was satisfied. The final learned reward bonus functions for each game were then evaluated in UCT play (using the same depth and trajectory parameters specified above) on 20 additional games, during which the reward function parameters were held fixed and UCT selected actions greedily according to the action value estimate of root nodes. Note that even though UCT chose actions according to action values greedily, there was still stochasticity in UCT's behavior because of the randomized tree expansion procedure of UCT.

[3] A discount factor of 0.9 to compute the accumulated discount sum of the first moment vector and 0.999 for the second order moment vector.

Table 3.1: Performance comparison of different UCT planners. The R^O columns denote UCT agents planning with objective rewards: R^O is depth 100 with 100 trajectories, R^O *(deeper)* is depth 200 with 100 trajectories, and R^O *(wider)* is depth 100 with 200 trajectories. The R^I column shows UCT's performance with internal rewards learned by PGRD-DL, with depth 100 and 100 trajectories. The table entries are mean scores over 20 evaluation games, and in parentheses are the standard errors of these means.

	Mean Game Score (standard error)			
Game	R^O	R^I	R^O *(deeper)*	R^O *(wider)*
Alien	2246 (139)	**12614** (1477)	2906 (387)	1795 (218)
Amidar	152 (13)	**1122** (139)	204 (20)	144 (16)
Assault	1477 (36)	1490 (32)	1495 (42)	**1550** (59)
Asterix	11700 (3938)	60353 (19902)	**99728** (16)	77211 (10377)
BankHeist	226 (13)	248 (13)	262 (17)	**284** (19)
BattleZone	8550 (879)	**17450** (1501)	13800 (1419)	8450 (1274)
BeamRider	2907 (322)	2794 (232)	**2940** (537)	2526 (333)
Berzerk	467 (25)	460 (26)	**506** (48)	458 (35)
Breakout	48 (14)	**746** (24)	516 (38)	79 (30)
Carnival	3824 (240)	**5610** (678)	3827 (173)	3553 (218)
Centipede	**4450** (236)	3987 (185)	2771 (231)	4076 (325)
DemonAttack	5696 (3316)	**121472** (201)	72968 (13590)	67166 (11604)
MsPacman	4928 (513)	**10312** (781)	6259 (927)	4967 (606)
Phoenix	5833 (205)	**6972** (371)	5931 (370)	6052 (330)
Pooyan	11110 (856)	**20164** (1015)	13583 (1327)	13106 (1605)
Q*Bert	2706 (409)	**47599** (2407)	6444 (1020)	4456 (688)
RiverRaid	3406 (149)	**5238** (335)	4165 (306)	4254 (308)
RoadRunner	8520 (3330)	**32795** (4405)	12950 (3619)	7217 (2758)
Robotank	2 (0.26)	3 (0.38)	**6** (0.84)	1 (0.33)
Seaquest	422 (19)	**2023** (251)	608 (41)	518 (45)
SpaceInvaders	1488 (114)	1824 (88)	**2154** (142)	1516 (166)
StarGunner	21050 (1507)	**826785** (3865)	33000 (4428)	22755 (1294)
UpNDown	127515 (10628)	103351 (5802)	109083 (9949)	**144410** (38760)
VideoPinball	702639 (17190)	736454 (23411)	**845280** (88556)	779624 (90868)
WizardOfWor	140475 (7058)	**198495** (225)	152886 (7439)	149957 (7153)

3.4.1 Learned Reward Bonuses Improve UCT

Table 3.1 shows the performance of UCT using the objective game-score-based reward (R^O column) and UCT with the learned reward bonus (R^I column). The values show the mean scores over 20 evaluation games and the numbers in parentheses are the standard errors of these means.

Table 3.2: Game score ratios of PGRD-DL to different UCT planners. The last two columns show the performance ratio of R^I compared to R^O and *max(R^O(deeper),R^O(wider))*. The R^I, R^O(deeper), and R^O(wider) agents take approximately equal time per decision.

	Mean Game Score Ratios	
Game	R^I / R^O	R^I / *max(R^O(deeper), R^O(wider))*
Alien	5.62	4.34
Amidar	7.39	5.50
Assault	1.01	0.96
Asterix	5.16	0.61
BankHeist	1.10	0.88
BattleZone	2.04	1.26
BeamRider	0.96	0.95
Berzerk	0.99	0.91
Breakout	15.47	1.45
Carnival	1.47	1.47
Centipede	0.90	0.98
DemonAttack	21.32	1.66
MsPacman	2.09	1.65
Phoenix	1.20	1.15
Pooyan	1.81	1.48
Q*Bert	17.59	7.39
RiverRaid	1.54	1.23
RoadRunner	3.85	2.53
Robotank	1.66	0.47
Seaquest	4.79	3.33
SpaceInvaders	1.23	0.85
StarGunner	39.28	25.05
UpNDown	0.81	0.72
VideoPinball	1.05	0.87
WizardOfWor	1.41	1.30

The results show that PGRD-DL learns reward bonus functions that improve UCT significantly on 18 out of 25 games. The mean scores of UCT with learned rewards are significantly higher than UCT using the game-score-based reward for these 18 games. The R^I/R^O column displays the ratio of the mean scores in column R^I to those in column R^O, and so a ratio over 1 implies improvement. These ratio values are plotted as a blue curve in Figure 3.1.

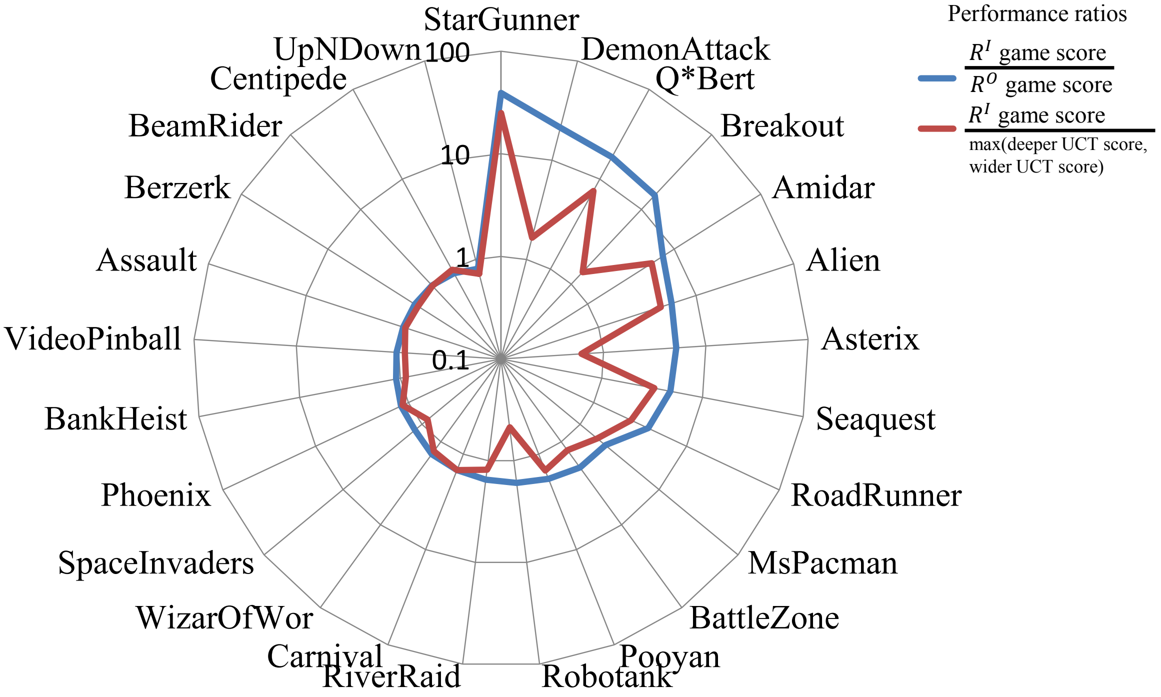


Figure 3.1: Performance comparison summary. The blue curve shows the ratio of the mean game score obtained by UCT planning with the PGRD-DL-adapted internal reward R^I, and the mean game score obtained by UCT planning with the objective reward R^O. The red curve shows the ratio of the mean game score obtained by UCT with R^I, and the mean game score of UCT planning with R^O with either deeper or more (wider) trajectories (whichever yields the highest score). UCT with the internal reward bonus outperforms the baseline if the ratio value lies outside the circle with radius 1. Games are sorted according to R^I/R^O.

3.4.2 Comparison Considering Computational Cost

The previous results establish that planning using the learned internal reward bonus can improve the performance of UCT on many ATARI games. But there is some computational overhead incurred in using the deep network to compute the internal reward bonuses. Is is worth spending the additional computational resources to compute the internal reward, as opposed to simply planning more deeply or broadly with the objective reward? We consider now the performance of two additional baselines that use additional computation to improve UCT without reward bonuses. The first baseline uses all the computational overhead in deeper planning. In this case, UCT plans to a depth of 200 frames instead of 100 frames (the number of trajectories is still 100 trajectories); we call this baseline *deeper UCT*. The second baseline uses all the computational overhead for more trajectories: UCT samples 200 trajectories to depth 100; we denote this baseline *wider UCT*. Both deeper UCT and wider UCT use about the same computational overhead as the UCT agent with reward bonuses that samples 100 trajectories to depth 100; the time per decision of deeper or wider UCT is slightly greater than UCT with reward bonuses (300 ms).

The mean scores of the deeper UCT and wider UCT are summarized in Table 3.1. We take the higher mean score of the deeper and wider UCTs as a useful assessment of performance obtainable using the computational overhead of reward bonuses for better planning, and compare it to the performance of UCT using the learned internal reward R^I. The last column in Table 3.2 displays the ratio of the mean games scores in column R^I to the better of wider UCT and deeper UCT, and this ratio appears as the red line in Figure 3.1. Among the 18 games in which reward bonuses improve UCT, reward bonuses outperform even the better of deeper or wider UCT agents in 15 games. These results show that the additional computational resources required to compute the reward bonuses may be better spent in this way than using those resources for more extensive planning.

3.4.3 The Nature of the Learned Reward-bonuses

What kinds of state and action discriminations has the reward bonus function learned? We consider now a simple summary visualization of the reward bonuses over time, as well as specific examples from the games Ms. Pacman and Q*Bert. The key conclusion from these analyses is that PGRD-DL learns useful game-specific features of state that help UCT planning, in part by mitigating the challenge of delayed reward.

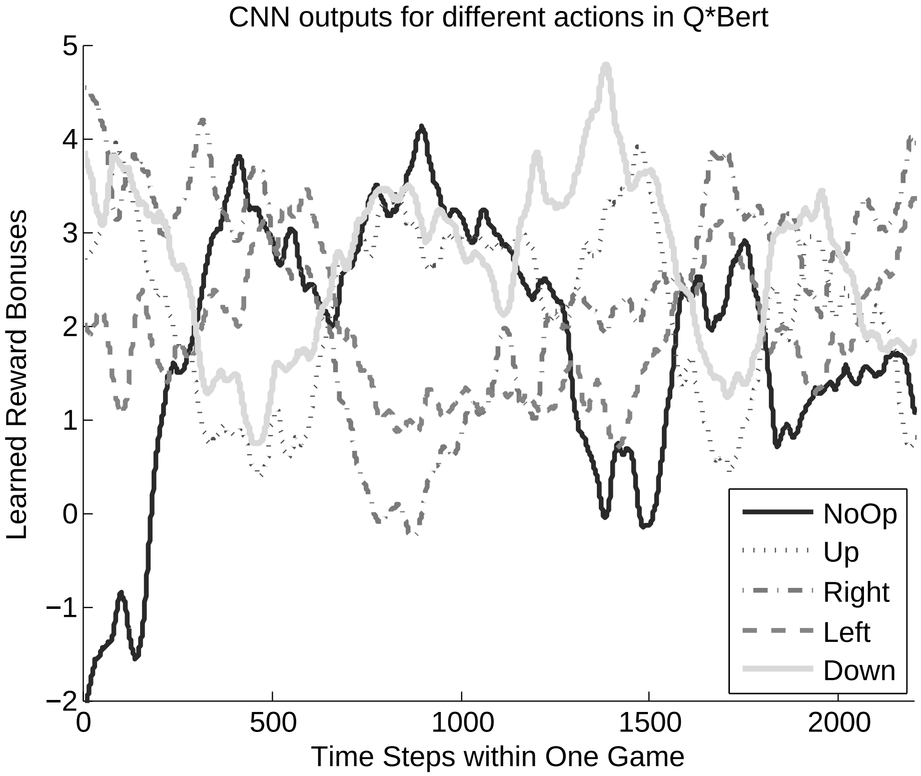
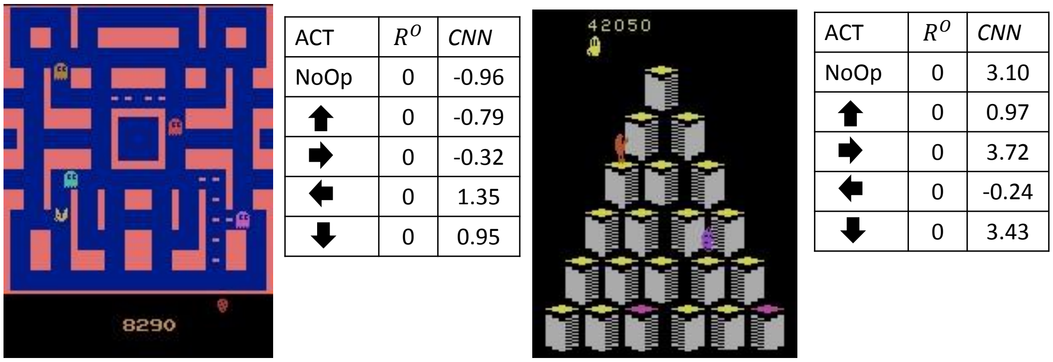


Figure 3.2: The learned reward bonuses for each of the five actions in Q*Bert for the states experienced during one game play. It is visually clear that different actions have the largest reward-bonus in different states.

Visualizing the dynamically changing reward bonus across states experienced in game play. Consider first how the learned reward bonus for each action changes as a function of state. Figure 3.2 shows the varying learned reward bonus values for each of the five actions in Q*Bert for the states experienced during one game play. The action with the highest (and lowest) reward bonus changes many times over the course of the game. The relatively fine-grained temporal dynamics of the reward bonuses throughout the game, and especially the change in relative ordering of the actions, provides support for the claim that the learned reward makes game-specific state discriminations—it is not simply unconditionally increasing or decreasing rewards for particular actions, which would have resulted in mostly flat lines across time in Figure 3.2. We now consider specific examples of the state discriminations learned.



ACT	R^O	*CNN*
NoOp	0	-0.96
⬆	0	-0.79
➡	0	-0.32
⬅	0	1.35
⬇	0	0.95

ACT	R^O	*CNN*
NoOp	0	3.10
⬆	0	0.97
➡	0	3.72
⬅	0	-0.24
⬇	0	3.43

Figure 3.3: Examples of how the bonus reward function represented by the CNN learns to encourage actions that avoid delayed bad outcomes. *Left*: A state in Ms. Pacman where the agent will encounter an enemy if it continues moving up. The learned reward bonus (under "CNN" in the small table) gives positive reward for actions taking the agent away and negative reward for actions that maintain course; the objective game score does not change (and so R^O is zero). *Right*: A state in Q*Bert where the agent could fall off the pyramid if it moves left and so left is given a negative bonus and other actions are given positive bonuses. The objective reward R^O is zero and indeed continues to be zero as the agent falls.

Examples of learned reward bonuses that capture delayed reward consequences. In the game Ms. Pacman, there are many states in which it is important to choose a movement direction that avoids subsequent encounters with enemies (and loss of a "Pacman life"). These choices may not yield differences in immediate reward and are thus examples of delayed reward consequences. Similarly, in Q*Bert, falling from the pyramid is a bad outcome but the falling takes many times steps before the "life" is lost and the episode ends. These consequences could in principle be taken into account by UCT planning with sufficient trajectories and depth. But we have discovered through observing the game play and examining specific bonus rewards that PGRD-DL learns reward bonuses that encourage action choices avoiding future enemy contact in Ms. Pacman and falling in Q*Bert (see Figure 3.3; the figure caption provides detailed descriptions.) The key lesson here is that PGRD-DL is learning useful and interesting game-specific state discriminations for a reward bonus function that mitigates the problem of delayed objective reward.

3.5 Conclusion

In this chapter we introduced a novel approach to combining Deep Learning and Reinforcement Learning by using the former to learn good reward-bonus functions from experience to improve the performance of UCT on ATARI games. Relative to the state-of-the art in the use of PGRD for reward design, we also provided the first example of automatically learning features of raw perception for use in the reward-bonus function, the first use of nonlinearly parameterized reward-bonus functions with PGRD, and provided empirical results on the most challenging domain of application of PGRD thus far. Our adaptation of PGRD uses a variance-reducing gradient procedure to stabilize the gradient calculations in the multi-layer CNN. Our empirical results showed that PGRD-DL learns reward-bonus functions that significantly improve the performance of UCT, and furthermore that the learned reward-bonus function mitigates the computational limitations in UCT in interesting ways. While our empirical results were limited to ATARI games, PGRD-DL is fairly general and we expect it to generalize it to other types of domains.

CHAPTER IV

Reward Transfer for Sequential Multiple Tasks

This chapter extends previous reward design work from single task scenarios into multiple sequential tasks. We consider agents that live for a long time in a sequential decision-making environment. While many different interpretations are possible for the notion of *long-lived*, here we consider agents that have to solve a sequence of tasks over a continuous lifetime. Thus, our problem is closely related to that of transfer learning in sequential decision-making, which can be thought of as a problem faced by agents that have to solve a set of tasks. Transfer learning (*Taylor and Stone* (2009)) has explored the *reuse* across tasks of many different components of a reinforcement learning (RL) architecture, including value functions (*Tanaka and Yamamura* (2003); *Konidaris and Barto* (2006); *Liu and Stone* (2006)), policies (*Natarajan and Tadepalli* (2005); *Torrey and Shavlik* (2010)), and models of the environment (*Atkeson and Santamaria* (1997); *Taylor et al.* (2008)). Other transfer approaches have considered parameter transfer (*Taylor et al.* (2007)), selective reuse of sample trajectories from previous tasks (*Lazaric et al.* (2008)), as well as reuse of learned abstract representations such as options (*Perkins and Precup* (1999); *Konidaris and Barto* (2007)).

A novel aspect of our transfer approach in long-lived agents is that we will reuse reward functions. At first blush, it may seem odd to consider using a reward function different from the one specifying the current task in the sequence (indeed, in most

RL research rewards are considered an immutable part of the task description). But there is now considerable work on designing good reward functions, including reward-shaping (*Ng et al.* (1999)), inverse RL (*Ng and Russell* (2000)), optimal rewards (*Singh et al.* (2010)) and preference-elicitation (*Chajewska et al.* (2000)). In this work, we specifically build on the insight of the optimal rewards problem (ORP; described in more detail in the next section) that guiding an agent's behavior with reward functions other than the task-specifying reward function can help overcome computational bounds in the agent architecture. We base our work on an algorithm from *Sorg et al.* (2010a) that learns good guidance reward functions incrementally in a single-task setting.

Our main contribution in this chapter is a new approach to transfer in long-lived agents in which we use good guidance reward functions learned on previous tasks in the sequence to incrementally train a *reward mapping function* that maps task-specifying reward functions into good initial guidance reward functions for subsequent tasks. We demonstrate that our approach can substantially improve a long-lived agent's performance relative to other approaches, first on an illustrative grid world domain, and second on a networking domain from prior work (*Natarajan and Tadepalli* (2005)) on the reuse of policies for transfer. In the grid world domain only the task-specifying reward function changes with tasks, while in the networking domain both the reward function and the state transition function change with tasks.

4.1 Related Work

We consider sequential decision-making environments formulated as controlled Markov processes (CMPs); these are defined via a state space S, an action space A, and a transition function T that determines a distribution over next states given a current state and action. A task in such a CMP is defined via a reward function R that maps state-action pairs to scalar values. The objective of the agent in a task is to execute the optimal policy, i.e., to choose actions in such a way as to optimize *utility* defined as the expected value of cumulative reward over some lifetime. A CMP and reward function together define a Markov decision process or MDP; hence tasks in this chapter are MDPs.

There are many approaches to planning an optimal policy in MDPs. Here we will use UCT (*Kocsis and Szepesvári* (2006)) which incrementally plans the action to take in the current state. It simulates a number of trajectories from the current state up to some maximum depth, choosing actions at each point based on the sum of an estimated action-value that encourages exploitation and a reward bonus that encourages exploration. It has theoretical guarantees of convergence and works well in practice on a variety of large-scale planning problems. We use UCT in this chapter because it is one of the state of the art algorithms in RL planning and because there exists a good optimal reward finding algorithm for it (*Sorg et al.* (2010a)).

Optimal Rewards Problem (ORP). In almost all of RL research, the reward function is considered part of the task specification and thus unchangeable. The optimal reward framework of *Singh et al.* (2010) stems from the observation that a reward function plays two roles simultaneously in RL problems. The first role is that of *evaluation* in that the task-specifying reward function is used by the agent designer to evaluate the actual behavior of the agent. The second is that of *guidance* in that the reward function is also used by the RL algorithm implemented by the agent to determine its behavior (e.g., via Q-learning (*Watkins and Dayan* (1992)) or UCT planning (*Kocsis and Szepesvári* (2006)). The optimal rewards problem separates these two roles into two separate reward functions, the task-specifying *objective* reward function used to evaluate performance, and an *internal* reward function used to guide agent behavior. Given a CMP M, an objective reward function R^o, an agent $\mathcal{A}$ parameterized by an internal reward function, and a space of possible internal reward functions $\mathcal{R}$, an optimal internal reward function R^{i^*} is defined as follows (throughout superscript o will denoted objective evaluation quantities and superscript i will denote internal quantities):

$$R^{i^*} = arg \max_{R^i \in \mathcal{R}} \mathbb{E}_{h \sim \langle \mathcal{A}(R^i), M \rangle} \Big\{ U^o(h) \Big\},$$

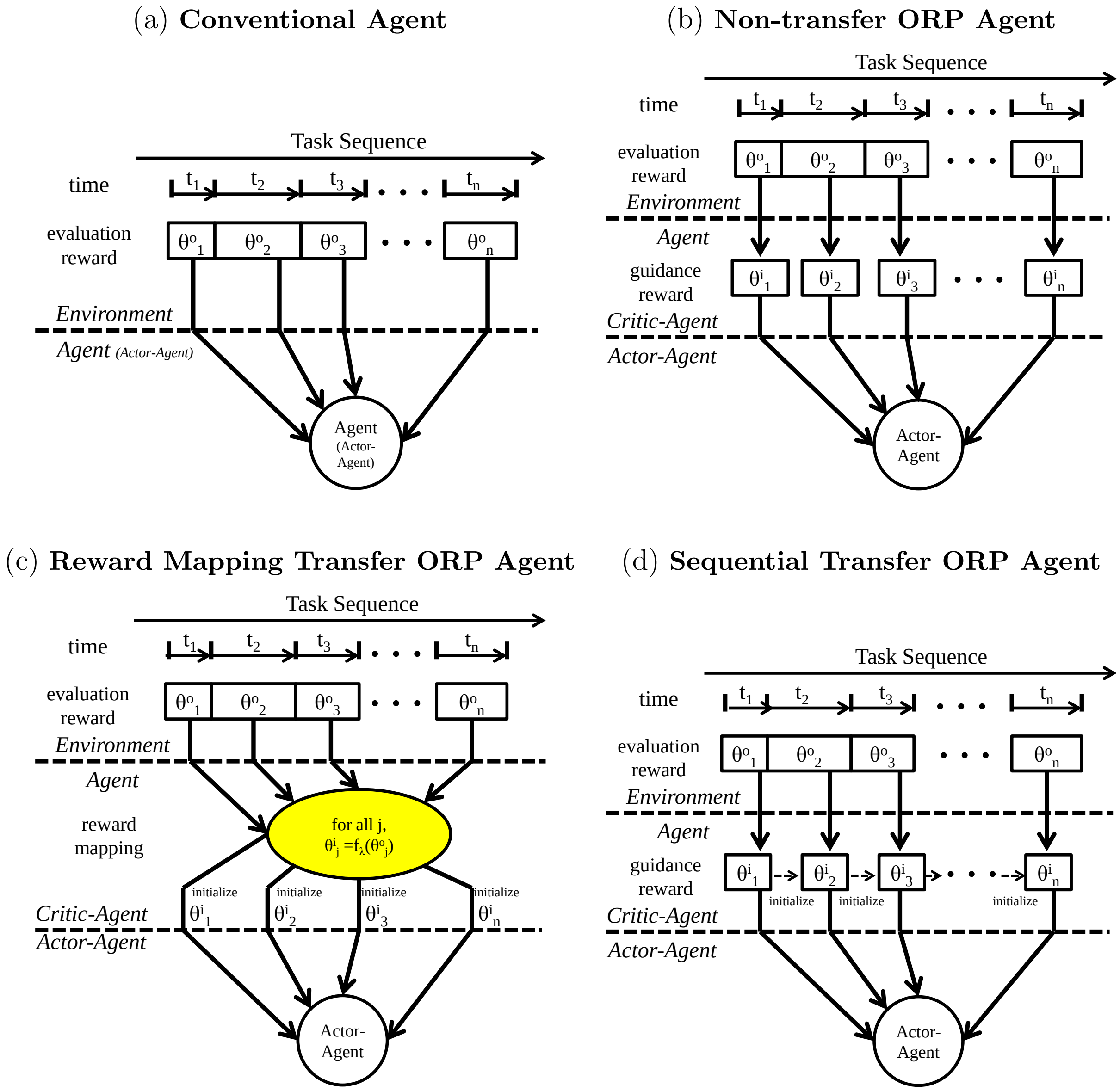
where $\mathcal{A}(R^i)$ is the agent with internal reward function R^i, $h \sim \langle \mathcal{A}(R^i), M \rangle$ is a random history (trajectory of alternating states and actions) obtained by the interaction of agent $\mathcal{A}(R^i)$ with CMP M, and $U^o(h)$ is the objective utility (as specified by R^o) to the agent designer of interaction history h. The optimal internal reward function will depend on the agent $\mathcal{A}$'s architecture and its limitations, and this distinguishes ORP from other reward-design approaches such as inverse-RL. When would the optimal internal reward function be different from the objective reward function? If an agent is *unbounded* in its capabilities with respect to the CMP then the objective reward function is always an optimal internal reward function. More crucially though, in the realistic setting of *bounded* agents, optimal internal reward functions may be quite different from objective reward functions. *Singh et al.* (2010) and *Sorg et al.* (2010a) provide many examples and some theory of when a good choice of internal reward can mitigate agent bounds, including bounds corresponding to limited lifetime to learn (*Singh et al.* (2010)), limited memory (*Sorg et al.* (2010a)), and limited resources for planning (the specific bound of interest in this chapter).

PGRD: Solving the ORP on-line while planning. Computing R^{i^*} can be computationally non-trivial. We will use *Sorg et al.* (2010a, 2011) policy gradient reward design (PGRD) method that is based on the insight that any planning algorithm can be viewed as procedurally translating the internal reward function R^i into behavior—that is, R^i are indirect parameters of the agent's policy. PGRD cheaply computes the gradient of the objective utility with respect to the R^i parameters through UCT planning. Specifically, it takes a simulation model of the CMP and an objective reward function and uses UCT to simultaneously plan actions with respect to the current internal reward function as well as to update the internal reward function in the direction of the gradient of the objective utility for use in the next planning step.

4.2 Four Agent Architectures for the Long-Lived Agent Problem

Long-Lived Agent's Objective Utility. We will consider the case where objective rewards are linear functions of objective reward features. Formally, the j^{th} task is defined by objective reward function $R^o_j(s,a) = \theta^o_j \cdot \psi^o(s,a)$, where θ^o_j is the parameter vector for the j^{th} task, ψ^o are the task-independent objective reward features of state and action, and '$\cdot$' denotes the inner-product. Note that the features are constant across tasks while the parameters vary. The j^{th} task lasts for t_j time steps. Given some agent $\mathcal{A}$ the expected objective utility achieved for a particular task sequence $\{\theta^o_j, t_j\}^K_{j=1}$, is $\mathbb{E}_{h\sim\langle\mathcal{A},M\rangle} \sum_{j=1}^K \left\{ U^{\theta^o_j}(h_j) \right\}$, where for ease of exposition we denote the history during task j simply as h_j. In general, there may be a distribution over task sequences, and the expected objective utility would then be a further expectation over such a distribution.

In some transfer or other long-lived agent research, the emphasis is on *learning* in that the agent is assumed to lack complete knowledge of the CMP and the task specifications. Our emphasis here is on *planning* in that the agent is assumed to know the CMP perfectly as well as the task specifications as they change. If the agent were unbounded in planning capacity, there would be nothing interesting left to consider because the agent could simply find the optimal policy for each new task and execute it. What makes our problem interesting therefore is that our UCT-based planning agent is computationally limited: the depth and number of trajectories



(a) **Conventional Agent**

(b) **Non-transfer ORP Agent**

(c) **Reward Mapping Transfer ORP Agent**

(d) **Sequential Transfer ORP Agent**

Figure 4.1: The four agent types compared in this chapter. In each figure, time flows from left to right. The sequence of objective reward parameters and task durations for n tasks are shown in the environment portion of each figure. In figures (b-d) the agent portion of the figure is further split into a critic-agent and an actor-agent; figure (a) does not have this split because it is the conventional agent. The critic-agent translates the objective reward parameters θ^o into the internal reward parameters θ^i. The actor-agent is a UCT agent in all our implementations. The critic-agent component varies across the figures and is crucial to understanding the differences among the agents (see text for detailed descriptions).

feasible are small enough (relative to the size of the CMP) that it cannot find near-optimal actions. This sets up the potential for both the use of the ORP and of transfer across tasks. Note that basic UCT does use a reward function but does not use an initial value function or policy and hence changing a reward function is a natural and consequential way to influence UCT. While non-trivial modifications of UCT could allow use of value functions and/or policies, we do not consider them here. In addition, in our setting a model of the CMP is available to the agent and so there is no scope for transfer by reuse of model knowledge. Thus, our reuse of reward functions may well be the most consequential option available in UCT.

Next we discuss four different agent architectures represented graphically in Figure 4.1, starting with a conventional agent that ignores both the potential of transfer and that of ORP, followed by three different agents that do not to varying degrees.

Conventional Agent. Figure 4.1(a) shows the baseline conventional UCT-based agent that ignores the possibility of transfer and treats each task separately. It also ignores ORP and treats each task's objective reward as the internal reward for UCT planning during that task.

The remaining three agents will all consider the ORP, and share the following details: The space of internal reward functions $\mathcal{R}$ is the space of all linear functions of internal reward features $\psi^i(s,a)$, i.e., $\mathcal{R}(s,a) = \{\theta \cdot \psi^i(s,a)\}_{\theta \in \Theta}$, where Θ is the space of possible parameters θ (in this chapter all finite vectors). Note that the internal reward features ψ^i and the objective reward features ψ^o do not have to be identical.

Non-Transfer ORP Agent. Figure 4.1(b) shows the non-transfer agent that ignores the possibility of transfer but exploits ORP. It initializes the internal reward function to the objective reward function of each new task as it starts and then uses PGRD to adapt the internal reward function while acting in that task. Nothing is transferred across task boundaries. This agent was designed to help separate the contributions of ORP and transfer to performance gains.

Reward-Mapping-Transfer ORP Agent. Figure 4.1(c) shows the reward-mapping agent that incorporates our main new idea. It exploits both transfer and ORP via incrementally learning a reward mapping function. A reward mapping function f maps objective reward function parameters to internal reward function parameters: $\forall j, \theta^i_j = f(\theta^o_j)$. The reward mapping function is used to initialize the internal reward function at the beginning of each new task. PGRD is used to continually adapt the initialized internal reward function throughout each task.

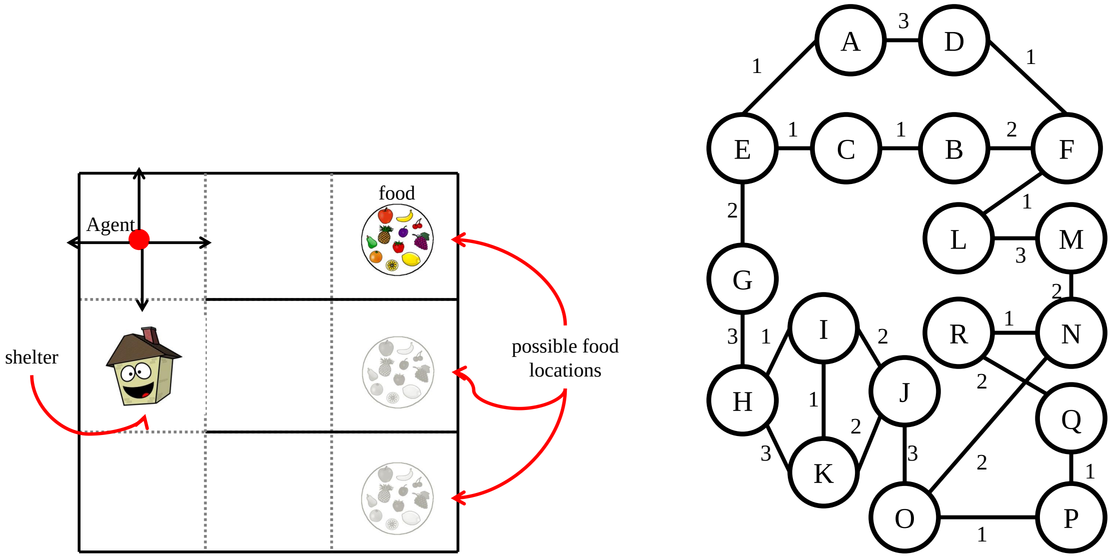
The reward mapping function is incrementally trained as follows: when task j ends, the objective reward function parameters θ^o_j and the adapted internal reward function parameters $\hat{\theta}^i_j$ are used as an input-output pair to update the reward mapping function. In our work, we use nonparametric kernel-regression to learn the reward mapping function. Pseudocode for a general reward mapping agent is presented in Algorithm 1.

Algorithm 1 General pseudocode for Reward Mapping Agent (Figure 1(c))

```
Input: $\{\theta^o_j, t_j\}_{j=1}^k$, where $j$ is task indicator, $t_j$ is task duration, and $\theta^o_j$ are the
objective reward function parameters specifying task $j$.

for $t = 1, 2, 3, ...$ do
    if a new task $j$ starts then
        obtain current objective reward parameters $\theta^o_j$
        compute: $\theta^i_j = f(\theta^o_j)$
        initialize the internal reward function using $\theta^i_j$
    end if
    $a_t$ := planning($s_t$; $\theta^i_j$) (select action using UCT guided by reward function $\theta^i_j$)
    ($s_{t+1}$, $r_{t+1}$) := takeAction($s_t$, $a_t$)
    $\theta^i$ := updateInternalRewardFunction($\theta^i$, $s_t$, $a_t$, $s_{t+1}$, $r_{t+1}$) (via PGRD)

    if current task ends then
        obtain current internal reward parameters as $\hat{\theta}^i_j$
        update reward mapping function $f$ using training pair ($\theta^o$, $\hat{\theta}^i_j$)
    end if
end for
```



(a) Food-and-Shelter Domain.

(b) Network Routing Domain.

Figure 4.2: Domains used in empirical evaluation; the network routing domain comes from (*Natarajan and Tadepalli* (2005)).

Sequential-Transfer ORP Agent. Figure 4.1(d) shows the sequential-transfer agent. It also exploits both transfer and ORP. However, it does not use a reward mapping function but instead continually updates the internal reward function across task boundaries using PGRD. The internal reward function at the end of a task becomes the initial internal reward function at the start of the next task achieving a simple form of sequential transfer.

4.3 Empirical Evaluation

The four agent architectures are compared to demonstrate that the reward mapping approach can substantially improve the bounded agent's performance, first on an illustrative grid world domain, and second on a networking routing domain from prior work (*Natarajan and Tadepalli* (2005)) on the transfer of policies.

4.3.1 Food-and-Shelter Domain

The purpose of the experiments in this domain are (1) to systematically explore the relative benefits of the use of ORP, and of transfer (with and without the use of the reward-mapping function), each in isolation and together, (2) to explore the sensitivity and dependence of these relative benefits on parameters of the long-lived setting such as mean duration of tasks, and (3) to visualize what is learned by the reward mapping function.

The environment is a simple 3 by 3 maze with three left-to-right corridors. Thick black lines indicate impassable walls. The position of the shelter and possible positions of food are shown in Figure 4.2.

Dynamics. The shelter breaks down with a probability of 0.1 at each time step. Once the shelter is broken, it remains broken until repaired by the agent. Food appears at the rightmost column of one of the three corridors and can be eaten by the agent when the agent is at the same location with the food. When food is eaten, new food reappears in a different corridor. The agent can move in four cardinal directions, and every movement action has a probability of 0.1 to result in movement in a random direction; if the direction is blocked by a wall or the boundary, the action results in no movement. The agent eats food and repairs shelter automatically whenever collocated with food and shelter respectively. The discount factor $\gamma = 0.95$.

State. A state is a tuple (l, f, h), where l is the location of the agent, f is the location of the food, and h indicates whether the shelter is broken.

Objective Reward Function. At each time step, the agent receives a positive reward of e (the eat-bonus) for eating food and a negative reward of b (the broken-cost) if the shelter is broken. Thus, the objective reward function's parameters are $\theta^o_j = (e_j, b_j)$, where $e_j \in [0, 1]$ and $b_j \in [-1, 0]$. Different tasks will require the agent to behave in different ways. For example, if (e_j, b_j) = (1,0), the agent should explore the maze to eat more food. If (e_j, b_j) = (0, -1), the agent should remain at the shelter's location in order to repair the shelter as it breaks.

Space of Internal Reward Functions. The internal reward function is $R^i_j(s) = R^o_j(s) + \theta^i_j \psi^i(s)$, where $R^o_j(s)$ is the objective reward function, $\psi^i(s) = 1 - \frac{1}{n_l(s)}$ is the inverse recency feature and $n_l(s)$ is the number of time steps since the agent's last visit to the location in state s. Since there is exactly one internal reward parameter, θ^i_j is a scalar. A positive θ^i_j encourages the agent to visit locations not visited recently, and a negative θ^i_j encourages the agent to visit locations visited recently.

Results: Performance advantage of reward mapping. 100 sequences of 200 tasks were generated, with Poisson distributions for task durations, and with objective reward function parameters sampled uniformly from their ranges. The agents used UCT with depth 2 and 500 trajectories; the conventional agent is thereby bounded as evidenced in its poor performance (see Figure 4.3).

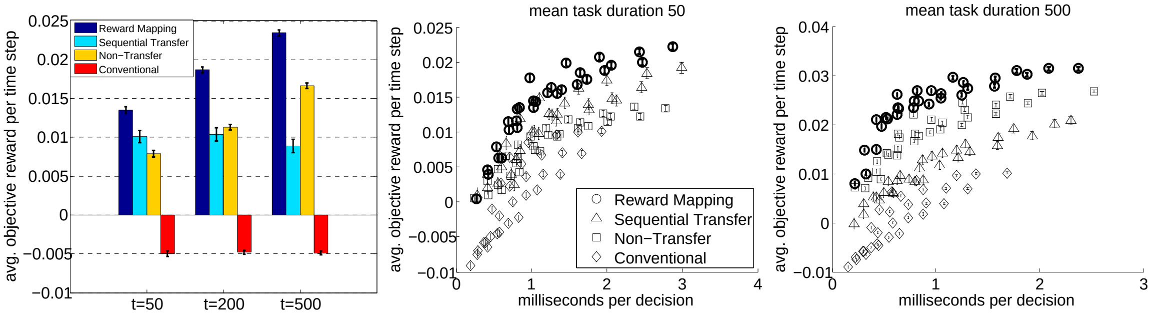


Figure 4.3: *(Left)* Performance of four agents in food-and-shelter domain at three different mean task durations. *(Middle and Right)* Comparing performance while accounting for computational overhead of learning and using the reward mapping function. See text for details.

The left panel in Figure 4.3 shows average objective reward per time step (with standard error bars). There are three sets of four bars each where each bar within a set is for a different architecture (see legend), and each set is for a different mean task duration (50, 200, and 500 from left to right). For each task duration the reward mapping agent does best and the conventional agent does the worst. These results demonstrate transfer helps performance and that transfer via the new reward mapping approach can substantially improve a bounded long-lived agent's performance relative to transfer via the competing method of sequential transfer. As task durations get longer the ratio of the reward-mapping agent's performance to the non-transfer agent's performance get smaller, though remains > 1 (by visually taking the ratio of the corresponding bars). This is expected because the longer the task duration the more time PGRD has to adapt to the task, and thus the less the better initialization provided by the reward mapping function matters.

In addition, the sequential transfer agent does better than the non-transfer agent for the shortest task duration of 50 while the situation reverses for the longest task duration of 500. This is intuitive and significant as follows. Recall that the initialization of the internal reward function from the final internal reward function of the previous task can hurt performance in the sequential transfer setting if the current task requires quite different behavior from the previous—but it can help if two successive tasks are similar. Correcting the internal reward function could cost a large number of steps. These effects are exacerbated by longer task durations because the agent then has longer to adapt its internal reward function to each task. In general, as task duration increases, the non-transfer agent improves but the sequential transfer agent worsens.

Results: Performance comparison considering computational overhead. The above results ignore the computational overhead incurred by learning and using the reward mapping function. The two rightmost plots in the bottom row of Figure 4.3 show the average objective reward per time step as a function of milliseconds per decision for the four agent architectures for a range of depth $\{1, \ldots, 6\}$, and trajectory-count $\{200, 300, \ldots, 600\}$ parameters for UCT. The plots show that for the entire range of time-per-decision, the best performing agents are reward-mapping agents—in other words, it is *not* better to spend the overhead time of the reward-mapping on additional UCT search. This can be seen by observing that the highest dot at any vertical column on the x-axis belongs to the reward mapping agent. Thus, the overhead of the reward mapping function in the reward mapping agent is insignificant relative to the computational cost of UCT (this last cost is all the conventional agent incurs).

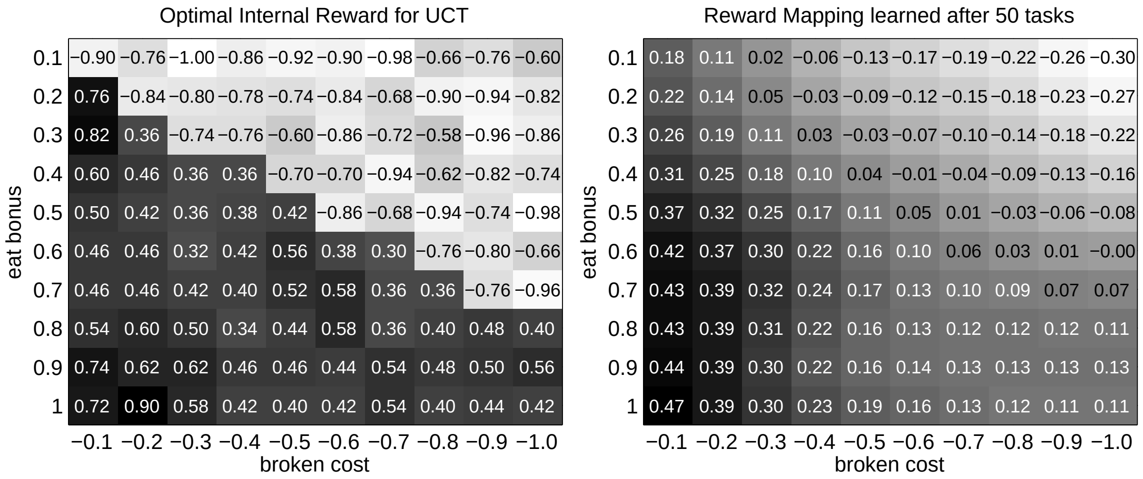


Figure 4.4: Reward mapping function visualization: *Top:* Optimal mapping, *Bottom:* Mapping found by the Reward Mapping agent after 50 tasks.

Results: Reward mapping visualization. Using a fixed set of tasks (as described above) with mean duration of 500, we estimated the optimal internal reward parameter (the coefficient of the inverse-recency feature) for UCT by a brute-force grid search. The optimal internal reward parameter is visualized as a function of the two parameters of the objective reward function (broken cost and eat bonus) in Figure 4.4, top. Negative coefficients (light color squares) for inverse-recency feature discourage exploration while positive coefficients (dark color squares) encourage exploration. As would be expected the top right corner (high penalty for broken shelter and low reward for eating) discourages exploration while the bottom left corner (high reward for eating and low cost for broken shelter) encourages exploration. Figure 4.4, bottom, visualizes the learned reward mapping function after training on 50 tasks. There is a clearly similar pattern to the optimal mapping in the upper graph, though it has not captured the finer details.

4.3.2 Network Routing Domain

The purposes of the following experiments are to (1) compare performance of our agents to a competing policy transfer method (*Natarajan and Tadepalli* (2005)) from a closely related setting on a networking application domain defined by the competing method; (2) demonstrate that our reward mapping and other agents can be extended to a multi-agent setting as required by this domain; and (3) demonstrate that the reward-mapping approach can be extended to handle task changes that involve changes to the transition function as well as objective reward.

The network routing domain (*Natarajan and Tadepalli* (2005)) (see Figure 4.2(b)) is defined from the following components. (1) A set of *routers*, or nodes. Every router has a queue to store packets. In our experiments, all queues are of size three. (2) A set of *links* between two routers. All links are bidirectional and full-duplex, and every link has a weight (uniformly sampled from {1,2,3}) to indicate the cost of transmitting a packet. (3) A set of active *packets*. Every packet is a tuple *(source, destination, alive-time)*, where *source* is the node which generated the packet, *destination* is the node that the packet is sent to, and *alive-time* is the time period that the packet has existed in the network. When a packet is delivered to its destination node, the alive-time is the end-to-end delay. (4) A set of *packet generators*. Every node has a packet generator that specifies a stochastic method to generate packets. (5) A set of *power consumption functions*. Every node's power consumption at time t is the number of packets in its queue multiplied by a scalar parameter sampled uniformly in the range $[0, 0.5]$.

Actions, dynamics, and states. Every node makes its routing decision separately and has its own action space (these determine which neighbor the first packet in the queue is sent to). If multiple packets reach the same node simultaneously, they are inserted into the queue in random order. Packets that arrives after the queue is full cause network congestion and result in packet loss. The global state at time t consists of the contents of all queues at all nodes at t.

Transition function. In a departure from the original definition of the routing domain, we parameterize the transition function to allow a comparison of agents' performance when transition functions change. Originally, the state transition function in the routing problem was determined by the fixed network topology and by the parameters of the packet generators that determined among other things the destination of packets. In our modification, nodes in the network are partitioned into three groups (G_1, G_2, and G_3) and the probabilities that the destination of a packet belongs to each group of nodes (p^{G_1}, p^{G_2}, and p^{G_3}) are parameters we manipulate to change the state transition function.

Objective reward function. The objective reward function is a linear combination of three objective reward features, the *delay* measured as the sum of the *inverse* end-to-end delay of all packets received at all nodes at time t, the *loss* measured as the number of lost packets at time t, and *power* measured as the sum of the power consumption of all nodes at time t. The weights of these three features are the parameters of the objective reward function. The weight for the delay feature $\in (0, 1)$, while the weights for both loss and power are $\in (-0.2, 0)$; different choices of these weights correspond to different objective reward functions.

Internal reward function. The internal reward function for the agent at node k is $R^i_{j,k}(s, a) = R^o_j(s, a) + \theta^i_{j,k}\psi^i_k(s, a)$, where $R^o_j(s, a)$ is the objective reward function, $\psi^i_k(s, a)$ is a binary feature vector with one binary feature for each (packet destination, action) pair. It sets the bits corresponding to the destination of the first packet in node k's queue at state s and action a to 1; all other bits are set to 0. The internal reward features are capable of representing arbitrary policies (and thus we also implemented classical policy gradient with these features using OLPOMDP (*Bartlett and Baxter* (2000)) but found it to be far slower than the use of PGRD with UCT and hence don't present those results here).

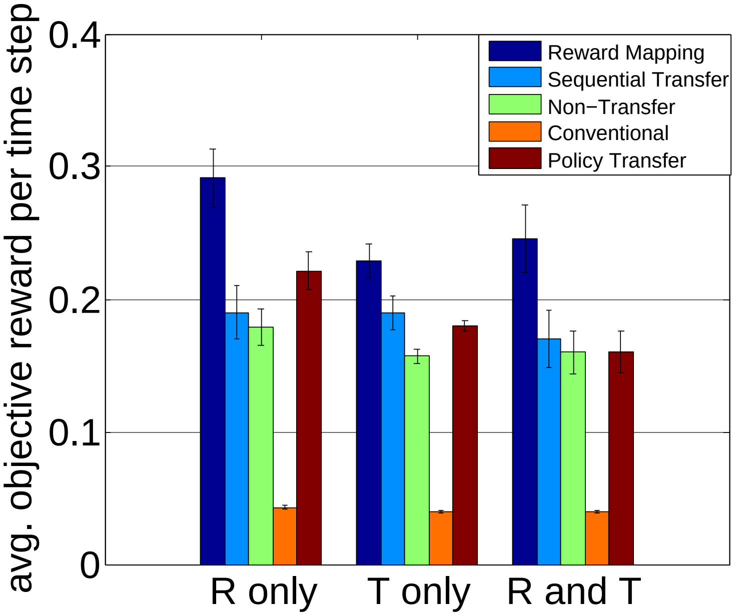


Figure 4.5: Performance on the network routing domain. (Left) tasks differ in objective reward functions (R) only. (Middle) tasks differ in transition function (T) only. (Right) tasks differ in both objective reward and transition (R and T) functions. See text for details.

Extension of Reward Mapping Agent to handle transition function changes. The parameters describing the transition function are concatenated with the parameters defining the objective reward function and used as input to the reward mapping function (whose output remains the initial internal reward function).

Handling Multi-Agency. Every nodes' agent observes the full state of the environment. All agents make decisions independently at each time step. Nodes do not know other nodes' policies, but can observe how the other nodes have acted in the past and use the empirical counts of past actions to sample other nodes' actions accordingly during UCT planning.

Competing policy transfer method. The competing policy transfer agent from *Natarajan and Tadepalli* (2005) reuses policy knowledge across tasks based on a model-based average-reward RL algorithm. Their method keeps a library of policies derived from previous tasks and for each new task chooses an appropriate policy from the library and then improves the initial policy with experience. Their policy selection criterion was designed for the case when only the linear reward parameters change. However, in our experiments, tasks could differ in three different ways: (1) only reward functions change, (2) only transition functions change, and (3) both reward functions and transition functions change. Their policy selection criterion is applied to cases (1) and (3). For case (2), when only transition functions change, their method is modified to select the library-policy whose transition function parameters are closest to the new transition function parameters.

Results: Performance advantage of Reward Mapping Agent. Three sets of 100 task sequences were generated, one in which the tasks differed in objective reward function only, another in which they differed in state transition function only, and third in which they differed in both. Figure 4.5 compares the average objective reward per time step for all four agents defined above as well as the competing policy transfer agent on the three sets. In all cases, the reward-mapping agent works best and the conventional agent worst. The competing policy transfer agent is second best when only the reward-function changes—just the setting for which it was designed.

4.4 Conclusion and Discussion

Reward functions are a particularly consequential locus for knowledge transfer; reward functions specify what the agent is to do but not how, and can thus transfer across changes in the environment dynamics (transition function) unlike previously explored loci for knowledge transfer such as value functions or policies or models. Building on work on the optimal reward problem for single task settings, our main algorithmic contribution for our long-lived agent setting is to take good guidance reward functions found for previous objective rewards and learn a mapping used to effectively initialize the guidance reward function for subsequent tasks. We demonstrated that our reward mapping approach can outperform alternate approaches; current and future work is focused on greater theoretical understanding of the general conditions under which this is true.

CHAPTER V

Learning to Query, Reason, and Answer Questions On Ambiguous Texts

This chapter addresses another source of rich perception, ambiguous texts, in reinforcement learning application. A key goal of research in conversational systems is to train an interactive agent to help a user with a task. Human conversation, however, is notoriously incomplete, ambiguous, and full of extraneous detail. To operate effectively, the agent must not only understand what was explicitly conveyed but also be able to reason in the presence of missing or unclear information. When unable to resolve ambiguities on its own, the agent must be able to ask the user for the necessary clarifications and incorporate the response in its reasoning. Motivated by this problem we introduce QRAQ ("crack"; Query, Reason, and Answer Questions), a new synthetic domain, in which a User gives an Agent a short story and asks a challenge question. These problems are designed to test the reasoning and interaction capabilities of a learning-based Agent in a setting that requires multiple conversational turns. A good Agent should ask only non-deducible, relevant questions until it has enough information to correctly answer the User's question. We use standard and improved reinforcement learning based memory-network architectures to solve QRAQ problems in the difficult setting where the reward signal only tells the Agent if its final answer to the challenge question is correct or not. To provide an upper-bound to the RL results we also train the same architectures using supervised information that tells the Agent during training which variables to query and the answer to the challenge question. We evaluate our architectures on four QRAQ dataset types, and scale the complexity for each along multiple dimensions.

In recent years, deep neural networks have demonstrated impressive performance on a variety of natural language tasks such as language modeling (*Mikolov et al.* (2010); *Sutskever et al.* (2011)), image captioning (*Vinyals et al.* (2015); *Xu et al.* (2015)), and machine translation (*Sutskever et al.* (2014); *Cho et al.* (2014); *Bahdanau et al.* (2014)). Encouraged by these results, machine learning researchers are now tackling a variety of even more challenging tasks such as reasoning and dialog. One such recent effort are the so-called "bAbI" problems of *Weston et al.* (2016). In these problems, the agent is presented with a short story and a challenge question that tests its ability to reason about the events in the story. The stories require the agent to figure out unstated constraints, but are otherwise self-contained, requiring no interaction between the agent and the environment. A very recent extension of this work (*Weston* (2016)) adds interaction by allowing the agent to respond in various ways to a teacher's questions.

There has also been significant recent interest in learning task-oriented dialog systems such as by *Bordes and Weston* (2016); *Dodge et al.* (2016); *Williams and Zweig* (2016); *Henderson et al.* (2014); *Young et al.* (2013). Here the agent is trained to help a user complete a task such as finding a suitable restaurant or movie. These tasks are typically modeled as slot-filling problems in which the agent knows about "slots", or attributes relevant to the task, and must determine which of the required slot values have been provided, querying the user for the others. For example, in a restaurant domain the slots might be: *location*, *price*, *cuisine*, etc. The reasoning required to decide on an action in these systems is primarily in determining which slot values the user has provided and which ones are required but still unknown to the agent. Realistic task-oriented dialog, however may require logical reasoning both to minimize irrelevant questions to the user and to focus the inquiry on questions most helpful to solving the user's task.

In this chapter we introduce a new simulator that generates problems in a domain we call QRAQ ("crack"; Query, Reason, and Answer Questions). In this domain the User provides a story and a challenge question to the agent, but with some of the entities replaced by variables. The introduction of variables, whose value may not be known, means that the agent must now learn additional challenging skills. First it must be able to decide whether it has enough information, in view of existing ambiguities, to answer the question. This requires reasoning about which variables can be deduced from other facts in the problem. Second, if the agent cannot answer the question by reasoning alone, it must learn to query the simulator for a variable value. To do this it must be able to infer which remaining variables are relevant to the question posed. The agent may be penalized for asking about irrelevant or deducible variables. Since there may be several rounds of questioning and reasoning, these requirements bring the problem closer to task-oriented dialog and represent a significant increase in the difficulty of the challenge over the original bAbI ("supporting fact") problems. In another significant departure from previous work on reasoning, including the work on the bAbI problems, we focus on the more realistic and challenging reinforcement learning (RL) setting in which the training agent is only told at the end of the multi-turn interaction whether its answer to the challenge question is correct or not. For an upper bound comparison, we also present the results of supervised training, in which we tell the agent which variable to query at each turn, and what to answer at the end.

In summary, this chapter presents two main contributions: (1) a novel domain, inspired by bAbI, but which additionally incorporates two crucial aspects of task-oriented dialog: *reasoning with incomplete information* and *interaction*, and (2) a baseline as well as an improved RL-based memory-network architecture with empirical results on our datasets that explore the robustness of the agent's reasoning.

5.1 Related Work

Our work builds on aspects of many different lines of machine learning research for which it is impossible to do full justice in the space available. Most relevant is research on deep neural networks and reinforcement learning for reasoning in natural language domains – in particular, those which make use of synthetic data.

One line of work which inspires our own is the development of novel neural architectures which can achieve deeper "understanding" of text input, thereby enabling more sophisticated reasoning and inference from source materials. In *Bordes et al.* (2010) for example, the model must integrate world knowledge to learn to label each word in a text with its "concept" which subsumes disambiguation tasks such as pronoun disambiguation. This is similar in spirit to our sub-task of deducing the value of variables but lacks the challenge of answering a question using this information or querying the user for more information. We draw particular inspiration for QRAQ from the bAbI problems of *Weston et al.* (2016) which are simple, automatically generated natural language stories, along with a variety of questions which can test many aspects of reasoning over the contents of such stories. The dynamic memory networks of *Kumar et al.* (2015) use the same synthetic domain and include tasks for both part-of-speech classification and question answering, but employ two Gated Recurrent Units (GRUs) to perform inference. Our problems subsume the reasoning required for the bAbI "supporting fact" tasks and add additional complexity in several ways. First, we allow ambiguous variables which require logical reasoning to deduce. Second, the question is not necessarily answerable with the information supplied and the agent must learn to decide if it has enough information to answer. Third, when the agent does not have the information it needs it must learn to query the user for a relevant fact and integrate the response into its reasoning. Such interactions may span multiple turns, essentially requiring a dialog.

There has been a lot of recent interest on the end-to-end training of dialog systems that are capable of generating a sensible response utterance at each turn, given the context of previous utterances in the dialog (*Vinyals and Le* (2015); *Serban et al.* (2016); *Lowe et al.* (2015); *Kadlec et al.* (2015); *Shang et al.* (2015)). Research on this topic tends to focus on large-scale training corpora such as movie subtitles, social media chats, or technical support logs. Because our problems are synthetic, our emphasis is not on the difficulties of understanding realistic language but rather on the mechanisms by which the reasoning and interaction process may be learned. For large corpora it is natural to use supervised training techniques where the Recurrent Neural Networks (RNNs) attempt to replicate the recorded human utterances. However, there are also approaches that envision training via reinforcement learning techniques, given a suitably defined reward function in the dialog (*Wen et al.* (2016); *Su et al.* (2015b,a)). We adopt this approach and similarly emphasize RL in this chapter.

As described in the previous Section, most of the reasoning emphasis in the slot-filling model of task-oriented dialogs is on understanding the user goal and which slot values have been given/which remain unfilled. By contrast, in QRAQ problems the emphasis is on inferring missing information if possible, and reasoning about what is important to ask, which can be much harder than evaluating which of the required slots are still unfilled. In more recent work on end-to-end learning of task-oriented dialog such as *Bordes and Weston* (2016); *Dodge et al.* (2016) this paradigm is extended to decompose the main task into smaller tasks each of which must be learned by the agent and composed to accomplish the main task. *Williams and Zweig* (2016) use an LSTM model that learns to interact with APIs on behalf of the user. *Weston* (2016) (bAbI-dialog) combines dialog and reasoning to explore how an agent can learn dialog when interacting with a teacher. The dataset is divided into

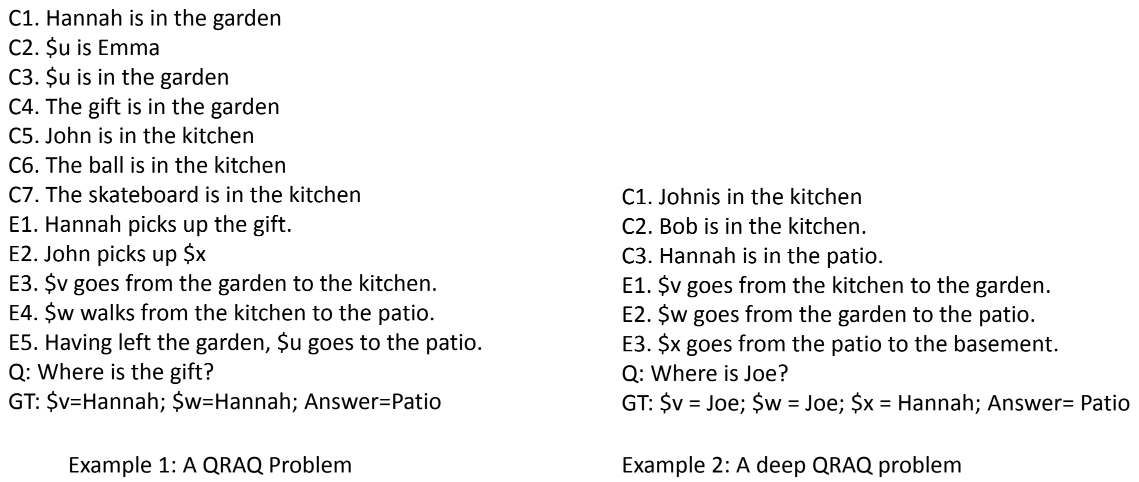
10 tasks, designed to test the ability of the agent to learn with different supervision regimes. The agent is asked a series of questions and given various forms of feedback according to the chosen scheme. For example: *imitating an expert*, *positive and negative feedback*, etc. Our focus is not on comparing supervision techniques, so instead we provide a numeric reward function for QRAQ problems that only assumes knowledge of when the agent's answer is right. In bAbI-dialog, the agent cannot ask questions, only answer them and receive guidance from the supervisor. The QRAQ problems by contrast allow the agent to query for the values of variables as well as to answer the challenge question. The information received must then be integrated into the reasoning process since the decision about what to query next (or answer) is dependent on the new state. In the bAbI-dialog problems the correct answer to a given question does not depend on previous questions and answers (though previous feedback can improve the agent's performance).

5.2 The QRAQ Domain

QRAQ problems have two actors, User and Agent. The User provides a short story, set in a domain similar to the HomeWorld domain of *Weston et al.* (2016); *Narasimhan et al.* (2015), and a *challenge question.* The stories are semantically coherent but may sometimes contain unknown variables, which the Agent may need to resolve to answer the question.[1]

Consider the simple QRAQ problem in Figure 5.1, where the context for the problem is labeled C1, C2, etc.; the events are labeled E1, E2, etc.; the question is labeled Q; and the ground truth is labeled GT. Here the entities $v and $w are

[1] Formally, the solution requires many unstated assumptions which the Agent must learn to correctly solve the problem. These include: domain constraints (e.g. a person can't be in two places at once), domain closure (the only entities in the world are those referenced explicitly in the story), closed world (the only facts true in the world are those given explicitly in the story), unique names (e.g. Emma != Hannah, kitchen != garden, etc.) and the frame assumption (the only things that change are those explicitly stated to change).



C1. Hannah is in the garden
C2. $u is Emma
C3. $u is in the garden
C4. The gift is in the garden
C5. John is in the kitchen
C6. The ball is in the kitchen
C7. The skateboard is in the kitchen
E1. Hannah picks up the gift.
E2. John picks up $x
E3. $v goes from the garden to the kitchen.
E4. $w walks from the kitchen to the patio.
E5. Having left the garden, $u goes to the patio.
Q: Where is the gift?
GT: $v=Hannah; $w=Hannah; Answer=Patio

Example 1: A QRAQ Problem

C1. Johnis in the kitchen
C2. Bob is in the kitchen.
C3. Hannah is in the patio.
E1. $v goes from the kitchen to the garden.
E2. $w goes from the garden to the patio.
E3. $x goes from the patio to the basement.
Q: Where is Joe?
GT: $v = Joe; $w = Joe; $x = Hannah; Answer= Patio

Example 2: A deep QRAQ problem

Figure 5.1: Examples of QRAQ data sets.

variables whose values are not provided to the Agent. In Event E3, for example, $v refers to either Hannah or Emma, but the Agent can't tell which. In a realistic text this might occur due to spelling or transcription errors, or indefinite pronouns such as "somebody". Variables such as $u (which aliases Emma) might realistically occur as descriptive references such as "John's sibling". The question to be answered "Where is the gift?" is taken to mean "Where is the gift at the end of the story?", a qualification which the Agent must learn. We call the entity referenced in the question (in this case "the gift") the *protagonist*.

The variables occurring in this example can be categorized as follows: A variable whose value can be determined by logical reasoning is called *deducible*; a variable which is consistently used to refer to an entity throughout the story is called an *alias*; a variable whose value is potentially required to solve the problem is called *relevant* (non-relevant variables are called *irrelevant*). Aliased variables may be defined in the context (e.g. $u is Emma).

In Figure 5.1, $x is irrelevant, since knowing its value doesn't help solve the problem. Similarly, we can observe that Emma (aliased as $u), leaves the garden in E5, making Hannah the only possible value of $v, so $v is deducible. Only $w

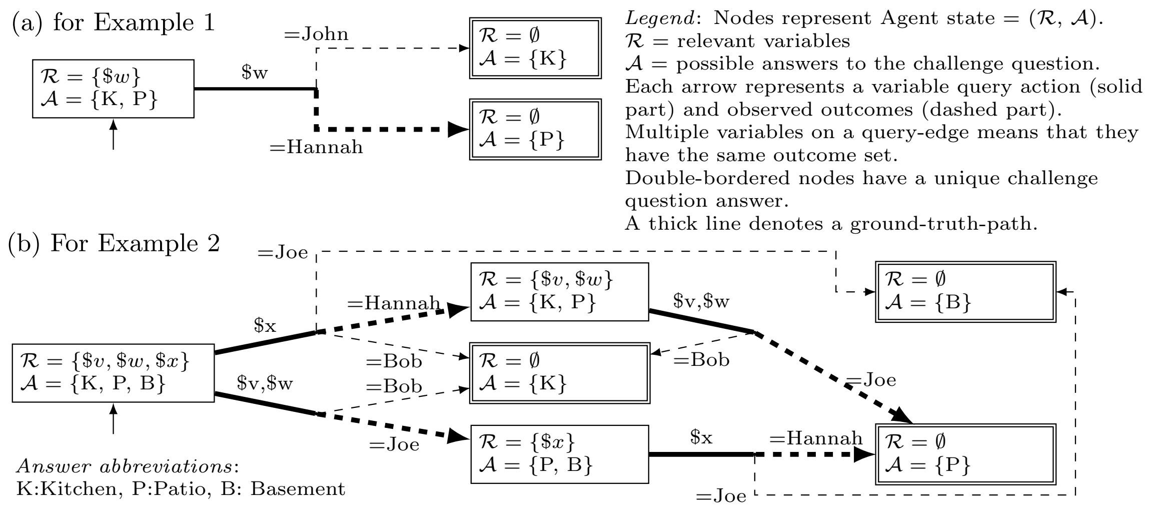

(a) for Example 1
$\mathcal{R} = \{\$w\}$
$\mathcal{A} = \{\text{K, P}\}$
\$w
=John
=Hannah
$\mathcal{R} = \emptyset$
$\mathcal{A} = \{\text{K}\}$
$\mathcal{R} = \emptyset$
$\mathcal{A} = \{\text{P}\}$
Legend: Nodes represent Agent state = ($\mathcal{R}$, $\mathcal{A}$).
$\mathcal{R}$ = relevant variables
$\mathcal{A}$ = possible answers to the challenge question.
Each arrow represents a variable query action (solid part) and observed outcomes (dashed part).
Multiple variables on a query-edge means that they have the same outcome set.
Double-bordered nodes have a unique challenge question answer.
A thick line denotes a ground-truth-path.
(b) For Example 2
$\mathcal{R} = \{\$v, \$w, \$x\}$
$\mathcal{A} = \{\text{K, P, B}\}$
\$x
\$v,\$w
=Joe
=Hannah
=Bob
=Bob
=Joe
$\mathcal{R} = \{\$v, \$w\}$
$\mathcal{A} = \{\text{K, P}\}$
$\mathcal{R} = \emptyset$
$\mathcal{A} = \{\text{K}\}$
$\mathcal{R} = \{\$x\}$
$\mathcal{A} = \{\text{P, B}\}$
\$v,\$w
=Bob
=Joe
\$x
=Hannah
=Joe
$\mathcal{R} = \emptyset$
$\mathcal{A} = \{\text{B}\}$
$\mathcal{R} = \emptyset$
$\mathcal{A} = \{\text{P}\}$
Answer abbreviations:
K:Kitchen, P:Patio, B: Basement

Figure 5.2: The QRAQ query graphs for both examples, illustrating the depth parameter.

remains as a relevant, non-deducible variable. To solve the problem, \$w must thus be queried by the Agent.

The deep example in Figure 5.1 shows a problem that requires two queries to solve. Here all the variables are relevant and none are deducible. Querying, say, \$v yields \$v = Joe. At this point, \$w becomes deducible and we can infer \$w = Joe. We now know that Joe is either in the patio or in the basement but not which. A further query for \$x reveals that \$x = Hannah, so the answer is that Joe is in the patio.

We can visualize the possible solutions to these problems using the "query graphs" shown in Figure 5.2. These graphs (technically DAGs) show the query policy required to solve the problem. Each node represents an informational state of the agent, and each edge the outcome of relevant queries. We define the *depth* of a query graph to be maximum number of variables that must be queried in the worst case[2], given the ground truth variable assignments. Examples of such paths in the graphs of Figure 5.2 are shown in bold. These paths pass only through states consistent with the ground truth. By this definition, Figure 5.2(a) has depth 1, and figure 5.2(b) has depth 2.

[2] In the best case the agent may get lucky and query a variable that yields the answer immediately.

As we discuss in Section 5.4.2, depth is an important driver of problem complexity.

There are many ways to solve these problems algorithmically. The QRAQ simulator uses a linear-time graph-based approach to track the possible locations of the protagonist as it processes the events. We will include a detailed description of the simulator and this algorithm when we release the QRAQ datasets to the research community.

5.3 Learning Agents

We develop and compare two different RL agents: (1) baseRL, which uses the memory network architecture from *Sukhbaatar et al.* (2015), and (2) impRL, which improves on the memory network architecture of baseRL by using a soft-attention mechanism over memory hops. We emphasize that the only feedback available to both agents, apart from a per-step penalty, is whether their answer to the challenge question is correct or not.

5.3.1 Control Loop

Both baseRL and impRL use the same control loop shown in Figure 5.3 whose steps are explained below:

(1) Initialization. For each problem, the challenge question is represented as a vector c, where c_i is the index of the i-th word in the question in a dictionary. The events and context sentences are encoded similarly and then the event vectors are appended to the context vectors to construct an initial memory matrix S_1, where S_1^{ij} is the index of the j-th word in the i-th sentence in the dictionary. Each word or variable comes from a global dictionary of N words formed by the union of all the words in the training and testing problems.

(2) Action Selection. The agent's policy at the t^{th} turn in the dialog is a function $\pi(a|S_t, c)$ mapping the memory matrix at turn t, S_t, and question, c, into a distribution over actions. An action, a, could be either a query for a variable or a final answer to the challenge question. The policy network, based on the end-to-end memory network, is shown in section 5.3.2.

(3) Variable Query and Memory Update. Whenever a query action is selected, the user module provides the true value for the corresponding variable in action a_t, i.e., the user module provides $v_t = \text{oracle}(a_t)$, where v_t is the dictionary index of the true word for the variable in action a_t. Then all occurrences of variable in action a_t in the memory S_t are replaced with the true value v_t: $S_{t+1} = S_t[a_t \rightarrow v_t]$. The new memory representation S_{t+1} is then used to determine the next action a_{t+1} in the next turn of the dialog.

(4) Final Answer Generation and Termination. If the action is an answer instead of a variable to query, the task terminates and a reward is generated based on whether the answer is correct or not (for the exact values of the rewards, see paragraph on "curriculum learning" in section 5).

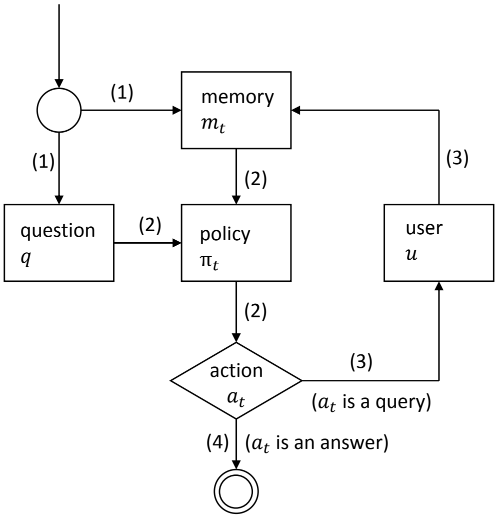


Figure 5.3: The control flow of the interaction of both our RL and SL architectures with the User.

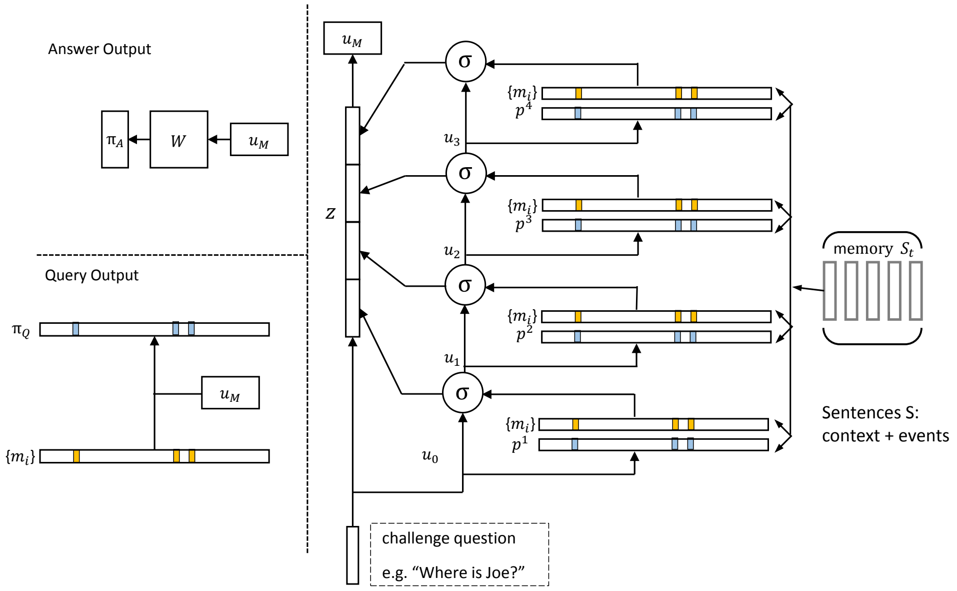


Figure 5.4: The impRL and impSL memory network architecture. See text for details.

5.3.2 baseRL: End-to-End Memory Network based Policy Learner

The baseRL agent builds on an end-to-end memory network policy as introduced by *Sukhbaatar et al.* (2015). It maps the memory matrix, S, and the challenge question representation, c, into an action distribution. Specifically, the i-th row of the memory matrix is encoded into a vector $m_i = \sum_j l_j \circ A[S^{ij}]$, where $A \in \mathbb{R}^{d \times N}$ is an embedding matrix and $A[k]$ returns the k-th column vector, d is the dimension of the embedding, '$\circ$' is an element wise multiplication operator, and l_j is a column vector with the structure $l_j^k = (1 - j/J) - (k/d)(1 - 2j/J)$ with J being the number of words in the sentences. Similarly, the challenge question is converted into a vector $q = \sum_j l_j \circ A[c_j]$.

The output vector upon addressing and then reading from $\{m_i\}$ in the k-th hop is u_k:

$$u_k = \tanh(H(o_k + u_{k-1})) \tag{5.1}$$

$$o_k = \sum_i p_i^k m_i, \tag{5.2}$$

$$p_i^k = \text{SoftMax}(u_{k-1}^\intercal m_i), \tag{5.3}$$

Where $\text{SoftMax}(z_i) = e^{z_i} / \sum_j e^{z_j}$ and $u_0 = q$. The policy module is implemented by two separate networks. One is for querying variables and the other is for answering the challenge questions.

Query network output The output of the query network is a distribution over variables in the memory matrix[3]. Since the problems have at most one variable per sentence, the distribution can be converted into the distribution over sentences without adding additional parameters. Specifically, if the i-th sentence has a variables, baseRL implements the following query policy:

$$\pi_Q^i = \text{SoftMax}(u_M^\intercal m_i) \tag{5.4}$$

where u_M is the output of the last memory hop and only sentences with variables are considered in the SoftMax computation.

Answer network output The output of the policy module is a distribution over potential actions $\{a_1, ..., a_K\}$. Specifically, baseRL implements the following policy:

$$\pi_A = \text{SoftMax}(Wu_M + b) \tag{5.5}$$

where u_M is the output of the last memory hop, $W \in \mathbb{R}^{K \times d}$ and $b \in \mathbb{R}^K$.

[3] A special query action is added to signal a switch from the query network to the answer network.

5.3.3 impRL: Improved End-to-End Memory Network based Policy Learner

In preliminary experiments with the baseRL architecture we tried optimizing the number of memory hops and discovered that the optimal number varied with each dataset. Rather than try to optimize it as a hyperparameter for each dataset, we developed a new architecture shown in Figure 5.4. Unlike the baseRL agent in which the final action-distribution output only conditions on the last hop output from the memory network architecture, the improved RL architecture, impRL, computes the final action-distribution-output over all memory hop outputs (with a fixed number of total hops across all datasets) as follows:

$$\pi_Q^i = \text{SoftMax}(u^\intercal m_i) \tag{5.6}$$

$$\pi_A = \text{SoftMax}(Wu + b) \tag{5.7}$$

$$u = \sum_j z_j u_j \tag{5.8}$$

$$z_j = \text{SoftMax}(q^\intercal u_j) \tag{5.9}$$

where $W \in \mathbb{R}^{K \times d}$ and $b \in \mathbb{R}^K$. Our modification to the memory network architecture of *Sukhbaatar et al.* (2015) is similar to the adaptive memory mechanism (AM) employed by *Weston et al.* (2016). The AM mechanism allows a variable number of memory hop computations in memory networks, but it is trained via extra supervision signals to determine when to stop memory hop computation explicitly. Our impRL mechanism departs from the AM mechanism in two ways: (1) it does not require extra supervision signals, and (2) while the AM mechanism uses the last memory hop's output for prediction, impRL instead uses a weighted average of all memory hops' outputs. As we show below impRL improves performance over baseRL in our empirical results for the more challenging datasets.

5.3.4 Policy Gradient & Reward Function

Much of the prior work on reasoning problems has used supervised learning (SL) methods. Here we focus on the far more realistic and challenging setting where the learning agent only has access to a binary reward function that evaluates whether the answer provided by the agent to the challenge question is correct or not. More specifically, we use a reward function that is positive when the action is the correct answer to the challenge question, and negative when the action is a wrong answer to the challenge question or the action is a query to a variable (see section 5.4 Curriculum Learning for the details). The penalty for a wrong answer is much larger than the penalty for querying a variable. Penalizing queries encourages the agent to query for the value of as few variables as possible. The objective function of the reinforcement learning method is to optimize the expected cumulative reward over (say M) training problem instances: $\sum_{m=1}^{M} \mathbb{E}\{\sum_t r_t^m\}$, where r_t^m is the immediate reward at the time step t of the m-th task. The GPOMDP algorithm with average reward baseline (*Weaver and Tao* (2001)) is used to calculate the policy gradient which in turn becomes the error signal to train all the parameters in both baseRL and impRL.

5.4 Data, Training, and Results

We evaluate our methods on four types of datasets described below. Each dataset contains 107,000 QRAQ problems, with 100,000 for training, 2,000 for testing, and 5,000 for validation.

(Loc). In this dataset, the context sentences describe people in rooms. The event sentences describe people (either by their name or by a variable) moving from room to room. The challenge questions are about the location of a specific person. We created several such datasets, scaling along vocabulary size, the number of sentences per problem (i.e., the sum of context and event sentences), the number of variables per problem and the depth of the problem. For a definition of "depth" see Section 5.2. For a detailed configuration of each data set see Table 5.1.

(+obj). This dataset adds objects to the Loc dataset. The context sentences describe people and objects in rooms. The event sentences describe people moving to and from various rooms, or picking up or dropping objects in rooms. People or objects (but not rooms) may be hidden by variables. The challenge questions are about the location of a person or an object.

(+alias). Adds aliases to the Loc dataset. Some of them are defined in the context.

(+par). This dataset modifies the Loc dataset by substituting sentences with semantically equivalent paraphrases. These paraphrases can change the number of words per sentence, and the ordering of the subject and object.

Pre-training The embedding matrix A is pre-trained via sentence level self-reconstruction. In the memory matrix S, each sentence i is represented as a row vector S^i and its corresponding embedding vector is $m_i = \sum_j l_j \cdot A[S^{ij}]$. The self-reconstruction objective is to maximize $\sum_i \sum_j \log(p_j(S^{ij}|m_i))$, where $p_j = \text{SoftMax}(W^{(j)}m_i)$ and $W \in \mathbb{R}^{N \times d}$. If a word position j has only one word candidate, then such a position is dropped from the objective function.

Time embedding for events If the i-th sentence is an event, a temporal embedding is added to its original embedding so that $m_i \leftarrow m_i + T(i)$, where $T(i) \in \mathbb{R}^d$ is the i-th column of a special matrix T that is learned from data.

Curriculum Learning We first encourage both reinforcement learning agents to query variables by assigning positive rewards for querying any variable. After convergence under this initial reward function, we switch to the true reward function that assigns a negative reward for querying a variable to reduce the number of unnecessary queries. Specifically, the rewards is +1 for correct final answers, -5 for wrong final

Table 5.1: Datasets. The first 7 rows give statistics on the datasets themselves. The last 8 rows show results for answer accuracy (AnsAcc), trajectory accuracy (TrajAcc), trajectory completeness (TrajCmpl) and query accuracy (QryAcc) for the impRL and baseRL agents on the respective datasets. The middle 8 rows show results for the supervised learning agents.

Data Set	*(Loc)*					*(+obj)*	*(+alias)*	*(+par)*
#names in vocab	5	20	10	20	20	20	20	20
#var in vocab	5	20	10	20	20	20	20	20
#sentence/prob.	5-6	5-6	7-10	15-20	19-23	7-10	10-12	10-12
#var/prob.	0-2	0-2	0-2	0-3	5-10	5-10	0-5	0
depth	0-2	0-2	0-2	0-2	4-9	0-2	0-5	0
avg. depth	0.817	0.872	0.558	0.459	5.087	0.543	1.066	-
sum(depth) / sum(#var)	0.734	0.748	0.313	0.204	0.703	0.404	0.310	-
AnsAcc in %; impSL	99.9	99.5	92.1	95.3	91.4	95.9	90.7	99.8
AnsAcc in %; baseSL	99.9	99.2	92.3	92.4	90.2	95.5	86.6	98.8
TrajAcc in %; impSL	99.6	98.9	90.2	88.4	85.3	95.2	86.7	-
TrajAcc in %; baseSL	98.9	98.7	90.3	86.5	83.3	94.9	85.3	-
TrajCmpl in %; impSL	99.5	98.8	89.9	85.6	80.9	94.9	83.6	-
TrajCmpl in %; baseSL	98.7	98.7	90.0	83.5	78.7	94.6	82.9	-
QryAcc in %; impSL	99.5	99.2	96.4	84.6	93.5	97.7	93.7	-
QryAcc in %; baseSL	98.7	99.3	96.3	85.5	92.7	97.5	97.0	-
AnsAcc in %; impRL	99.1	94.4	86.5	89.0	64.2	81.1	75.7	96.9
AnsAcc in %; baseRL	98.4	95.0	88.4	88.2	54.6	79.6	69.7	97.2
TrajAcc in %; impRL	94.5	90.9	61.9	52.0	45.1	74.9	63.2	-
TrajAcc in %; baseRL	94.8	90.4	63.6	52.5	35.7	73.9	60.5	-
TrajCmpl in %; impRL	94.5	88.7	55.8	46.9	37.8	61.8	56.4	-
TrajCmpl in %; baseRL	94.6	89.5	59.9	47.4	28.3	61.2	54.5	-
QryAcc in %; impRL	94.3	95.4	49.2	32.1	80.0	69.6	77.0	-
QryAcc in %; baseRL	95.5	94.1	54.6	32.0	76.5	71.0	79.6	-

answers, and the reward for query is initially +0.05 then changed to -0.05.

Action Mask When choosing an action (query-variable or answer) from the policy network, output units that correspond to choices *not* available in the set of sentences at that turn[4] in the dialog are not considered in the selection process.

Exploration To encourage exploration, which is needed in RL settings, an exploration policy π' is defined, given the agent's learned policy π as follows. A random

[4] When a variable is queried, the simulation engine replaces every occurrence of a variable with its value and returns the updated text to the agent, at the beginning of the next turn.

action is chosen with probability ϵ and the remaining $1 - \epsilon$ probability is distributed over actions as $\pi'(a) = (\pi(a) + \delta)/(1 + |A|\delta)$, where $\delta = \epsilon/(1-\epsilon)|A|$, and $|A|$ is the number of actions. For our experiments, $\epsilon = 0.1$.

We used ADAM (*Kingma and Ba* (2015)) to optimize the parameters of the networks. The number of memory hops is fixed to 4. The embedding dimensionality is fixed to 50.

5.4.1 Supervised Learning Baselines: baseSL and impSL

We trained the architecture of baseRL and impRL using supervised learning signals to provide a ceiling (i.e. an upper-bound) on achievable performance to understand the relative performance of the reinforcement learning agents. In the supervised learning setting, the relevant variables and (when appropriate) the correct final answers are provided at each turn, and the cross entropy loss function is used to optimize the parameters.

5.4.2 Results & Analysis

We compute the following metrics on the test set for each QRAQ dataset.

Let $\mathcal{T}$ be the set of all trajectories (a trajectory is the sequence of variable queries ending in an answer) produced by the agent on the test problems, $\mathcal{T}_A$ be the set of trajectories where the agent gave the correct answer, $\mathcal{T}_C$ be the set of trajectories completed without any remaining relevant variables, and let $\mathcal{T}_Q$ be the set of trajectories completed without ever querying an irrelevant variable (our QRAQ simulator tracks these statistics). Let $\mathcal{N}_Q$ be the number of queried variables and $\mathcal{N}_{relQ}$ be the

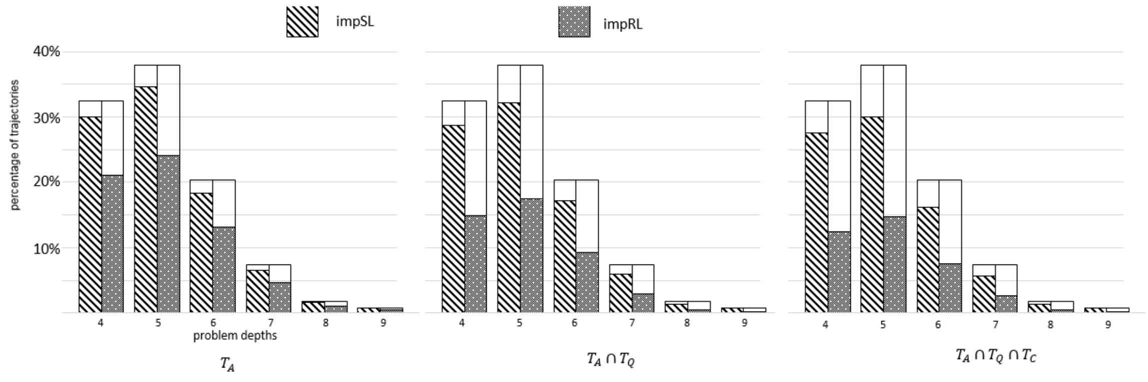
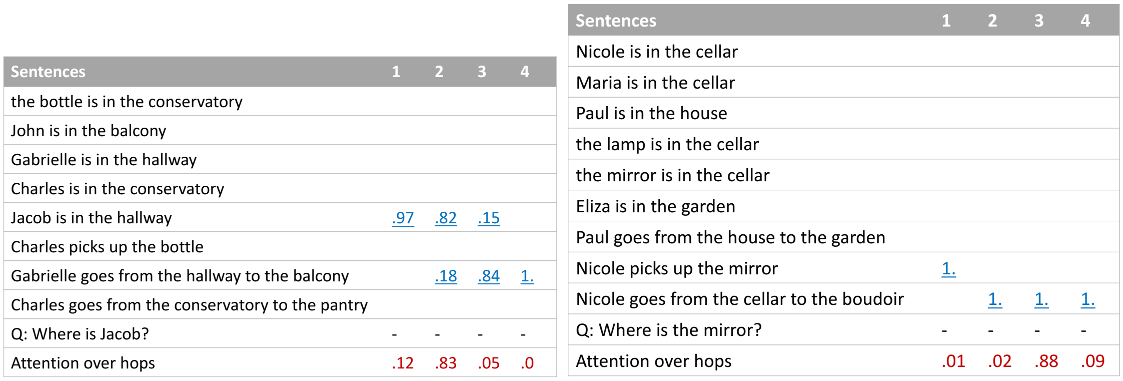


Figure 5.5: Test trajectory distributions over problem depths for impSL and impRL on the hardest Loc dataset. The bars show the the percentage of trajectories satisfying corresponding criteria for impSL and impRL. The figures show that impSL performs much better than impRL on deeper problems.

number of queried variables that were relevant. Then

$$\begin{aligned}
\text{answer-accuracy} &= \frac{|\mathcal{T}_A|}{|\mathcal{T}|} \\
\text{trajectory-completeness} &= \frac{|\mathcal{T}_A \cap \mathcal{T}_Q \cap \mathcal{T}_C|}{|\mathcal{T}|} \\
\text{trajectory-accuracy} &= \frac{|\mathcal{T}_A \cap \mathcal{T}_Q|}{|\mathcal{T}|} \\
\text{query-accuracy} &= \frac{\mathcal{N}_{relQ}}{\mathcal{N}_Q}.
\end{aligned}$$

By comparing the various metrics in the rows of Table 5.1 we can make a number interesting observations. First, the performance gap between the supervised learning agents and reinforcement learning agents increases as problems become more complex. Figure 5.5 shows the test trajectory distributions over problem depths. The result shows that impSL performs much better than impRL on deeper problems. This indicates that the RL agent is sensitive to the scaling and hasn't learned as robust an algorithm as the supervised agent (which, considering the stronger training signal for the supervised agent, was to be expected). Second, for all reinforcement learning agents, the answer accuracy is considerably higher than the trajectory accuracy. This



Sentences	1	2	3	4
the bottle is in the conservatory				
John is in the balcony				
Gabrielle is in the hallway				
Charles is in the conservatory				
Jacob is in the hallway	.97	.82	.15	
Charles picks up the bottle				
Gabrielle goes from the hallway to the balcony		.18	.84	1.
Charles goes from the conservatory to the pantry				
Q: Where is Jacob?	-	-	-	-
Attention over hops	.12	.83	.05	.0

Sentences	1	2	3	4
Nicole is in the cellar				
Maria is in the cellar				
Paul is in the house				
the lamp is in the cellar				
the mirror is in the cellar				
Eliza is in the garden				
Paul goes from the house to the garden				
Nicole picks up the mirror	1.			
Nicole goes from the cellar to the boudoir		1.	1.	1.
Q: Where is the mirror?	-	-	-	-
Attention over hops	.01	.02	.88	.09

Figure 5.6: Visualization of attentions of the trained impRL. The underlined (blue) scores in a column show the attention values over sentences within the memory hop. Our impRL has four memory hops and each memory hop's values are shown in the corresponding column. The (red) scores in the bottom line show the attention values over memory hops. The attention over memory hops is able to softly select the output of different hops.

indicates that answering is considerably easier for the agent than solving the whole problem perfectly, with only relevant queries followed by a correct answer. Third, in the simplest (Loc) datasets (the leftmost 3 to 4 columns) where the number of sentences, number of variables, and depth are all small, both baseRL and impRL do well and furthermore do similarly well. This is also the case for the +obj dataset because it is also similarly simple (in terms of depth range). As expected the answer accuracy, the trajectory accuracy and trajectory completeness get worse as the problems get more complex (in terms of parameters listed in the first 7 rows). This decrease in performance is roughly seen left to right in the 5 columns for the (Loc) datasets. Next, note that the query accuracy results of the leftmost 5 columns closely track the ratio of depth to the number of variables in the problems. That ratio is a rough estimate of the percentage of relevant variables to the total number of variables. An agent who guessed which variables were relevant would be sensitive to this ratio, doing better when it was high and worse when low, lending weight to the hypothesis that the RL agent's query algorithm is underperforming. Of all the parameters explored, the 'depth' and '#sentence/prob' parameters seem the most impactful. Specifically there is a sharp drop-off in performance for both the base and improved architectures in the rightmost (Loc) column where the depth is 4-9 compared with 0-2 in the leftmost four (Loc) columns. Similarly the (+obj) dataset has a low depth and the performance is similarly good. Finally, we note that in the more complex data sets including the rightmost column of the (Loc) set of columns as well as for the (+alias) dataset the performance of both baseRL and impRL is worse than for the simpler datasets but in all these cases impRL improves the answer accuracy significantly. The attention over hops in our improved architecture allows soft selection of different hop outputs as shown in Figure 5.6.

5.5 Conclusion

We have introduced the new QRAQ domain that presents an agent with a story consisting of context sentences, temporally ordered event sentences with variables, and a challenge question. Answering the challenge question requires multi-turn interactions in which a good agent should ask only non-deducible and relevant questions at any turn. We presented and evaluated two RL-based memory network architectures, a baseline and an improved architecture in which we added soft-attention over multiple memory hops. Our results show that that both architectures do solve these challenging problems to substantial degrees, even on the quite unforgiving trajectory accuracy measure, and despite being limited to feedback only about whether the the answer is correct. At the same time, as the gap between the supervised and RL approaches shows, there is considerable room for innovation in the reinforcement learning setting for this domain.

CHAPTER VI

Discussion and Future Work

This dissertation aims to address the challenges posed by the interaction of rich perception and delayed rewards in reinforcement learning domains. There are two fairly recent and quite exciting developments that form the departure point of this dissertation. The first is the idea of combining RL methods with Deep Learning methods as a practical means for addressing the rich perception challenges, and the second is the reward design approach to address the delayed rewards for the state-of-the-art Monte Carlo tree search methods. The four projects in the dissertation explore these two ideas and make substantial steps towards addressing the challenges posed by the rich perception and delayed reward issues. We summarize our contributions in Section 6.1, and suggest future work in Section 6.2.

6.1 Contributions of this Dissertation

Deep Learning for Real-time Control using Offline Monte Carlo Tree Search

The main contribution of Chapter II is the idea to use a small amount of data collected from a slow UCT-based planning agent to train a fast deep convolutional neural network for real-time control. UCT has the nice theory property that the number of simulation steps needed to ensure any bound on the loss of following the UCT policy is independent of the size of the state space. The results capture the fact that the use of UCT avoids the perception problem and UCT is suitable for large state space sequential decision problems, but UCT is unrealistic in real time control because it requires substantial computation for every time step of action selection. The idea of Chapter II combines the performance advantage (in terms of expected cumulative rewards) of the slow UCT planners and the real-time control capability of deep neural networks by training the deep neural networks' outputs to approximate the output values/policies of the slow UCT planners. The main applied results in Chapter II show that such approach outperforms the classic Deep Q-Network in several Atari games. Such approach is also applicable to other real-time control tasks as long as the task environment simulators are available, such as robot navigations.

Deep Learning for Reward Design to Improve Monte Carlo Tree Search

The key contribution of Chapter III is a new end-to-end reward design algorithm to mitigate the delayed rewards for the state-of-the-art MCTS method. The reward design algorithm converts visual perceptions into reward bonuses via deep neural networks, and optimizes the network weights to improve the performance of MCTS end-to-end via policy gradient. Relative to the state-of-the-art in the use of PGRD for reward design, Chapter III provided the first example of automatically learning features of raw perception for use in the reward-bonus function, the first use of non-linearly parameterized reward-bonus functions with PGRD, and the empirical results on the most challenging domain of application of PGRD thus far. The adaptation of PGRD uses a variance-reducing gradient procedure to stabilize the gradient calculations in the multi-layer CNN. The empirical results show that PGRD-DL learns reward-bonus functions that significantly improve the performance of UCT. While the empirical results were limited to ATARI games in Chapter III, PGRD-DL is fairly general and we expect to generalize it to other types of RL planners and domains.

Reward Transfer for Sequence Multiple Tasks

Chapter IV considers reward functions for knowledge transfer in task sequence scenarios. The main contribution is the idea to learn a reward mapping function that maps task parameters to reward bonus parameters to transfer reward bonus for UCT planners in task sequence. Reward functions specify what the agent is to do but not how, and can thus transfer across changes in the environment dynamics (transition function) unlike previously explored components for knowledge transfer such as value functions or policies. Building on work on the optimal reward problem for single task settings, the main algorithmic contribution for the task sequence setting is to take good guidance reward functions found for previous objective rewards and learn a mapping used to effectively initialize the guidance reward function for subsequent tasks. The empirical results demonstrate that the reward mapping approach can outperform alternate approaches. The reward transfer idea could be applied to other more complex tasks with the help of more flexible reward mapping functions, such as deep neural networks.

Learning to Query, Reason, and Answer Questions on Ambiguous Texts

Chapter V has two contributions. The first one is a new QRAQ domain to evaluate RL agents' ability in reasoning with information uncertainty. The data set presents a RL agent with a story consisting of context sentences, temporally ordered event sentences with variables, and a challenge question. Answering the challenge question requires multi-turn interactions in which a good RL agent should ask only non-deducible and relevant questions at any turn. The second contribution is to show memory network based RL agents could solve the tasks to substantial degree. Two RL-based memory network architectures were presented and evaluated, a baseline and an improved architecture in which we added soft-attention over multiple memory hops. The results show that both architectures do solve these challenging problems to substantial degrees, even on the quite unforgiving trajectory accuracy measure, and despite being limited to feedback only about whether the the answer is correct. Even though the RL methods were only evaluated in the synthetic data domains, we believe that the work presented in this chapter is in the direction towards real-world reasoning and dialog applications.

6.2 Future Work

The dissertation closes with a few future work to improve on the existing algorithmic ideas in the chapters.

Policy Distillation for Roll-out Policy Learning

Chapter II demonstrated that we could use a small amount of data collected from a slow UCT-based planning agent to train a fast deep neural network for real-time control. However, the trained neural network is never applied back into UCT planning. One potential improvement is to replace the roll-out random policies in the UCT planning procedure with the trained fast neural networks. After the replacement, the return values of the roll-out policy would be more informative and also have less variance. The performance of the UCT planner could be substantially improved. We can collect data from the improved UCT planner to train new neural networks, and continue this loop to train even stronger neural networks for real-time control.

Joint Learning of Reward Bonuses and Roll-out Policies

The reward design algorithm in Chapter III uses the execution trajectories of UCT planners to learn end-to-end reward bonus functions to improve UCT performance. Besides rewards in planning, roll-out policies also have a huge impact on the performance of the UCT planners. The execution trajectories could also be used to train a better roll-out policy as described in *Policy Distillation for Roll-out Policy Learning.* The joint learning of roll-out policies and reward bonuses could make a better use of the execution trajectories of the UCT planners.

End-to-End Memory-based Reward Design

Memory or frequency based hand-crafted features are proven to be particularly useful in previous reward design work. These features could help the agent explore the environment in a systematic way. The convolutional neural networks used in Chapter III are not able to replicate previous memory or frequency based features. One potential extension in learning more flexible reward bonus function is to introduce memory based neural network architectures into the reward bonus functions.

Learning When to Use Reward Bonuses

The use of reward bonus functions mitigates the delayed rewards in Monte Carlo Tree Search methods, but at a cost of additional computational overheads. The design of the neural network based reward bonus function faces a dilemma. Deep and sophisticated neural networks are required to provide sufficient expressiveness for the reward bonuses, but shallow and simple neural networks are preferred because of computational efficiency. One potential direction to address this dilemma is to learn when to use the sophisticated reward bonus functions. Be more specific, we could train a computationally cheap controller whose binary output determines whether to invoke the reward bonus function for a root node or a subtree. The computationally cheap controller could be trained together with the reward bonus functions via policy gradient procedures.

BIBLIOGRAPHY

BIBLIOGRAPHY

Asmuth, J., M. L. Littman, and R. Zinkov (2008), Potential-based shaping in model-based reinforcement learning, in *Proceedings of the 23rd national conference on Artificial intelligence.*

Atkeson, C. G., and J. C. Santamaria (1997), A comparison of direct and model-based reinforcement learning, in *International Conference on Robotics and Automation*, pp. 3557–3564.

Bahdanau, D., K. Cho, and Y. Bengio (2014), Neural machine translation by jointly learning to align and translate, *arXiv preprint arXiv:1409.0473.*

Bartlett, P. L., and J. Baxter (2000), Stochastic optimization of controlled partially observable markov decision processes, in *Proceedings of the 39th IEEE Conference on Decision and Control.*, vol. 1, pp. 124–129.

Bellemare, M., J. Veness, and M. Bowling (2012a), Sketch-based linear value function approximation, in *Advances in Neural Information Processing Systems.*

Bellemare, M., J. Veness, and M. Bowling (2013a), Bayesian learning of recursively factored environments, in *Proceedings of the 30th International Conference on Machine Learning.*

Bellemare, M. G., J. Veness, and M. Bowling (2012b), Investigating contingency awareness using ATARI 2600 games, in *the 26th AAAI Conference on Artificial Intelligence.*

Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling (2013b), The Arcade Learning Environment: an evaluation platform for general agents, *Journal of Artificial Intelligence Research.*

Bellemare, M. G., J. Veness, and E. Talvitie (2014), Skip context tree switching, in *Proceedings of the International Conference on Machine Learning.*

Bengio, Y. (2009), Learning deep architectures for AI, *Foundations and trends in Machine Learning.*

Bordes, A., and J. Weston (2016), Learning end-to-end goal-oriented dialog, *arXiv preprint arXiv:1605.07683.*

Bordes, A., N. Usunier, R. Collobert, and J. Weston (2010), Towards understanding situated natural language, *Proc. of AISTATS*.

Buro, M. (2002), Improving heuristic mini-max search by supervised learning, *Artificial Intelligence*, *134*(1), 85–99.

Chajewska, U., D. Koller, and R. Parr (2000), Making rational decisions using adaptive utility elicitation, in *Proceedings of the 17th National Conference on Artificial Intelligence*.

Childs, B. E., J. H. Brodeur, and L. Kocsis (2008), Transpositions and move groups in Monte Carlo tree search, in *IEEE Symposium On Computational Intelligence and Games*.

Cho, K., B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014), Learning phrase representations using rnn encoder–decoder for statistical machine translation, in *Proceedings of the International Conference for Learning Representations*.

Ciresan, D., U. Meier, and J. Schmidhuber (2012), Multi-column deep neural networks for image classification, in *IEEE Conference on Computer Vision and Pattern Recognition*.

Dodge, J., A. Gane, X. Zhang, A. Bordes, S. Chopra, et al. (2016), Evaluating prerequisite qualities for learning end-to-end dialog systems, *Proceedings of the International Conference for Learning Representations*.

Erhan, D., Y. Bengio, A. Courville, and P. Vincent (2009), Visualizing higher-layer features of a deep network, *Tech. rep.*, University of Montreal.

Gelly, S., and D. Silver (2011), Monte Carlo tree search and rapid action value estimation in computer Go, *Artificial Intelligence*.

Graves, A., A. Mohamed, and G. Hinton (2013), Speech recognition with deep recurrent neural networks, in *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649.

Guo, X., S. Singh, and R. L. Lewis (2013), Reward mapping for transfer in long-lived agents, in *Advances in Neural Information Processing Systems*, pp. 2130–2138.

Guo, X., S. Singh, H. Lee, R. L. Lewis, and X. Wang (2014), Deep learning for real-time ATARI game play using offline Monte-Carlo tree search planning, in *Advances in Neural Information Processing Systems*.

Guo, X., S. Singh, R. Lewis, and H. Lee (2016), Deep learning for reward design to improve monte carlo tree search in atari games, *Proc. International Joint Conference on Artificial Intelligence*.

Hannun, A., et al. (2014), Deep speech: Scaling up end-to-end speech recognition, *arXiv preprint arXiv:1412.5567.*

Hausknecht, M., and P. Stone (2015), Deep recurrent q-learning for Partially Observable MDPs, in *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents.*

Hausknecht, M., P. Khandelwal, R. Miikkulainen, and P. Stone (2012), HyperNEAT-GGP: A hyperNEAT-based Atari general game player, in *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, pp. 217–224.

Henderson, M., B. Thomson, and S. Young (2014), Word-based dialog state tracking with recurrent neural networks, *Proc. of SIGDIAL-2014.*

Hostetler, J., A. Fern, and T. Dietterich (2014), State aggregation in Monte Carlo tree search, in *the 28th AAAI Conference on Artificial Intelligence.*

Jiang, N., S. Singh, and R. Lewis (2014), Improving UCT planning via approximate homomorphisms, in *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems.*

Kadlec, R., M. Schmid, and J. Kleindienst (2015), Improved deep learning baselines for ubuntu corpus dialogs, in *Proc. of NIPS-15 Workshop on "Machine Learning for SLU and Interaction".*

Karpathy, A., G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei (2014), Large-scale video classification with convolutional neural networks, in *IEEE Conference on Computer Vision and Pattern Recognition.*

Kearns, M., Y. Mansour, and A. Y. Ng (2002), A sparse sampling algorithm for near-optimal planning in large Markov decision processes, *Machine Learning*, *49*(2-3), 193–208.

Kingma, D., and J. Ba (2015), Adam: A method for stochastic optimization, in *Proceedings of the 3rd International Conference for Learning Representations.*

Kocsis, L., and C. Szepesvári (2006), Bandit based monte-carlo planning, in *the 15th European Conference on Machine Learning.*

Konidaris, G., and A. Barto (2006), Autonomous shaping: Knowledge transfer in reinforcement learning, in *Proceedings of the 23rd International Conference on Machine learning*, pp. 489–496.

Konidaris, G., and A. G. Barto (2007), Building portable options: Skill transfer in reinforcement learning, in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, vol. 2, pp. 895–900.

Koutník, J., G. Cuccu, J. Schmidhuber, and F. Gomez (2013), Evolving large-scale neural networks for vision-based reinforcement learning, in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012), ImageNet classification with deep convolutional neural networks, in *Advances in Neural Information Processing Systems*.

Kumar, A., O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, and R. Socher (2015), Ask me anything: Dynamic memory networks for natural language processing, *arXiv preprint arXiv:1506.07285*.

Lazaric, A., M. Restelli, and A. Bonarini (2008), Transfer of samples in batch reinforcement learning, in *Proceedings of the 25th International Conference on Machine learning*, pp. 544–551.

Le, Q. V., W. Y. Zou, S. Y. Yeung, and A. Y. Ng (2011), Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis, in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3361–3368.

Le, Q. V., M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng (2012), Building high-level features using large scale unsupervised learning, in *Proceedings of the 29th International Conference on Machine Learning*.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998), Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, *86*(11), 2278–2324.

Lee, H., R. Grosse, R. Ranganath, and A. Y. Ng (2009), Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations, in *Proceedings of the 26th Annual International Conference on Machine Learning*.

Liu, Y., and P. Stone (2006), Value-function-based transfer for reinforcement learning using structure mapping, in *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, vol. 21(1), p. 415.

Lowe, R., N. Pow, I. Serban, and J. Pineau (2015), The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems, in *Proc. of SIGDIAL-2015*.

Mikolov, T., M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur (2010), Recurrent neural network based language model, in *Eleventh Annual Conference of the International Speech Communication Association*.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller (2013), Playing Atari with deep reinforcement learning, in *Deep Learning, Neural Information Processing Systems Workshop*.

Mnih, V., K. Kavukcuoglu, D. Silver, and et al. (2015), Human-level control through deep reinforcement learning, *Nature.*

Mohamed, A., G. E. Dahl, and G. Hinton (2012), Acoustic modeling using deep belief networks, *IEEE Transactions on Audio, Speech, and Language Processing*, *20*(1), 14–22.

Narasimhan, K., T. Kulkarni, and R. Barzilay (2015), Language understanding for text-based games using deep reinforcement learning, *arXiv preprint arXiv:1506.08941.*

Natarajan, S., and P. Tadepalli (2005), Dynamic preferences in multi-criteria reinforcement learning, in *Proceedings of the 22nd International Conference on Machine learning.*

Ng, A. Y., and S. J. Russell (2000), Algorithms for inverse reinforcement learning, in *Proceedings of the 17th International Conference on Machine Learning.*

Ng, A. Y., D. Harada, and S. Russell (1999), Policy invariance under reward transformations: Theory and application to reward shaping, in *Proceedings of the 16th International Conference on Machine Learning.*

Oh, J., X. Guo, H. Lee, R. Lewis, and S. Singh (2015), Action-conditional video prediction using deep networks in ATARI games, in *Advances in Neural Information Processing Systems.*

Perkins, T. J., and D. Precup (1999), Using options for knowledge transfer in reinforcement learning, *University of Massachusetts, Amherst, MA, USA, Tech. Rep.*

Rosin, C. D. (2011), Nested rollout policy adaptation for monte carlo tree search, in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, AAAI Press.

Ross, S., G. J. Gordon, and J. A. Bagnell (2011), A reduction of imitation learning and structured prediction to no-regret online learning, in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics.*

Schaeffer, J., J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron (1992), A world championship caliber checkers program, *Artificial Intelligence*, *53*(2), 273–289.

Schmidhuber, J. (2015), Deep learning in neural networks: An overview, *Neural Networks.*

Schulman, J., S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel (2015), Trust region policy optimization, in *Proceedings of the 32nd International Conference on Machine Leanring.*

Serban, I. V., A. Sordoni, Y. Bengio, A. Courville, and J. Pineau (2016), Building end-to-end dialogue systems using generative hierarchical neural network models, in *Proceedings of the 30th AAAI Conference on Artificial Intelligence*.

Shang, L., Z. Lu, and H. Li (2015), Neural responding machine for short-text conversation, in *Proc. of ACL-2015*.

Shannon, C. E. (1950), Programming a computer for playing chess, *Philosophical Magazine*.

Sheppard, B. (2002), World-championship-caliber scrabble, *Artificial Intelligence*, *134*(1), 241–275.

Silver, D., et al. (2016), Mastering the game of go with deep neural networks and tree search, *Nature*, *529*(7587), 484–489.

Singh, S., R. L. Lewis, A. G. Barto, and J. Sorg (2010), Intrinsically motivated reinforcement learning: An evolutionary perspective, *IEEE Transactions on Autonomous Mental Development.*, *2*(2), 70–82.

Sorg, J., R. L. Lewis, and S. Singh (2010a), Reward design via online gradient ascent, in *Advances in Neural Information Processing Systems*.

Sorg, J., S. Singh, and R. L. Lewis (2010b), Internal rewards mitigate agent boundedness, in *Proc. 27th International Conference on Machine Learning*.

Sorg, J., S. Singh, and R. L. Lewis (2011), Optimal rewards versus leaf-evaluation heuristics in planning agents, in *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence*.

Srinivasan, S., E. Talvitie, and M. Bowling (2015), Improving exploration in UCT using local manifolds, in *Proceedings of the 26th Conference on Artificial Intelligence*.

Stadie, B. C., S. Levine, and P. Abbeel (2015), Incentivizing exploration in reinforcement learning with deep predictive models, *arXiv preprint arXiv:1507.00814*.

Su, P.-H., D. Vandyke, M. Gasic, D. Kim, N. Mrksic, T.-H. Wen, and S. Young (2015a), Learning from real users: Rating dialogue success with neural networks for reinforcement learning in spoken dialogue systems, in *Proc. of INTERSPEECH-2015*.

Su, P.-H., D. Vandyke, M. Gasic, N. Mrksic, T.-H. Wen, and S. Young (2015b), Reward shaping with recurrent neural networks for speeding up on-line policy learning in spoken dialogue systems, *Proc. of SIGDIAL-2015*.

Sukhbaatar, S., J. Weston, R. Fergus, et al. (2015), End-to-end memory networks, in *Advances in Neural Information Processing Systems*.

Sutskever, I., J. Martens, and G. E. Hinton (2011), Generating text with recurrent neural networks, in *Proceedings of the 28th International Conference on Machine Learning*, pp. 1017–1024.

Sutskever, I., O. Vinyals, and Q. V. Le (2014), Sequence to sequence learning with neural networks, in *Advances in neural information processing systems*, pp. 3104–3112.

Sutton, R., and A. G. Barto (1998), *Reinforcement learning: An introduction*, MIT press.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2015), Going deeper with convolutions, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9.

Szepesvári, C. (2010), Algorithms for reinforcement learning, *Synthesis lectures on artificial intelligence and machine learning*, *4*(1), 1–103.

Tanaka, F., and M. Yamamura (2003), Multitask reinforcement learning on the distribution of mdps, in *Proceedings IEEE International Symposium on Computational Intelligence in Robotics and Automation.*, vol. 3, pp. 1108–1113.

Taylor, M. E., and P. Stone (2009), Transfer learning for reinforcement learning domains: A survey, *The Journal of Machine Learning Research*, *10*, 1633–1685.

Taylor, M. E., S. Whiteson, and P. Stone (2007), Transfer via inter-task mappings in policy search reinforcement learning, in *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, p. 37.

Taylor, M. E., N. K. Jong, and P. Stone (2008), Transferring instances for model-based reinforcement learning, in *Machine Learning and Knowledge Discovery in Databases*, pp. 488–505.

Tesauro, G. (1994), TD-Gammon, a self-teaching backgammon program, achieves master-level play, *Neural computation.*

Tesauro, G. (1995), Temporal difference learning and TD-gammon, *Communications of the ACM*, *38*(3), 58–68.

Torrey, L., and J. Shavlik (2010), Policy transfer via Markov logic networks, in *Inductive Logic Programming*, pp. 234–248, Springer.

van Hasselt, H., A. Guez, and D. Silver (2016), Deep reinforcement learning with double q-learning, in *Thirtieth AAAI Conference on Artificial Intelligence.*

Veness, J., D. Silver, A. Blair, and W. Uther (2009), Bootstrapping from game tree search, in *Advances in neural information processing systems*, pp. 1937–1945.

Vinyals, O., and Q. Le (2015), A neural conversational model, *Proceedings of the International Conference on Machine Learning, Workshop.*

Vinyals, O., A. Toshev, S. Bengio, and D. Erhan (2015), Show and Tell: A neural image caption generator, in *IEEE Conference on Computer Vision and Pattern Recognition*.

Watkins, C. J., and P. Dayan (1992), Q-learning, *Machine learning*, *8*(3-4), 279–292.

Weaver, L., and N. Tao (2001), The optimal reward baseline for gradient-based reinforcement learning, in *Proceedings of the 17th conference on Uncertainty in Artificial Intelligence*.

Wen, T.-H., M. Gasic, N. Mrksic, L. M. Rojas-Barahona, P.-H. Su, S. Ultes, D. Vandyke, and S. Young (2016), A network-based end-to-end trainable task-oriented dialogue system.

Weston, J. (2016), Dialog-based language learning, *arxiv preprint arXiv:1604.06045*.

Weston, J., A. Bordes, S. Chopra, and T. Mikolov (2016), Towards AI-complete question answering: A set of prerequisite toy tasks, *Proceedings of the International Conference for Learning Representations*.

Williams, J. D., and G. Zweig (2016), End-to-end lstm-based dialog control optimized with supervised and reinforcement learning, *arXiv preprint arXiv:1606.01269*.

Xu, K., J. Ba, R. Kiros, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio (2015), Show, Attend and Tell: Neural image caption generation with visual attention, in *Proceedings of the International Conference on Machine Learning*.

Young, S., M. Gasic, B. Thomson, and J. D. Williams (2013), Pomdp-based statistical spoken dialog systems: A review, *Proceedings of the IEEE*, *101*(5), 1160–1179.