UNIVERSITY OF CALIFORNIA, MERCED

# Learning Representations in Reinforcement Learning

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

**Jacob Rafati Heravi**

Committee in charge:

Professor David C. Noelle, Chair
Professor Marcelo Kallmann
Professor Roummel F. Marcia
Professor Shawn Newsam
Professor Jeffrey Yoshimi

2019

# Copyright Notice

The Dissertation of Jacob Rafati Heravi is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

---

Marcelo Kallmann

---

Roummel F. Marcia

---

Shawn Newsam

---

Jeffrey Yoshimi

---

David C. Noelle, Chair

University of California, Merced

2019

# Dedication

To my mother. She has worked so hard, and has supported me throughout my entire life all alone. She has tolerated our 8,000 mile distance for over 5 years due to the current travel bans. I hope this accomplishment brings joy to her life.

To Katie Williams, for all the emotional, and spiritual support, and for all love.

# Contents

# List of Symbols

| | |
|---|---|
| $\alpha$ | Learning rate |
| $\mathcal{D}$ | Agent's experience memory |
| $\mathcal{D}_1$ | Controller's experience memory in HRL |
| $\mathcal{D}_2$ | Meta-controller's experience memory in HRL |
| $\epsilon$ | Exploration rate in $\epsilon$-greedy method |
| $\mathcal{G}$ | Subgoals space |
| $\gamma$ | Discount factor |
| $\mathcal{L}$ | Objective function |
| $\mathcal{A}$ | Actions space |
| $\mathcal{S}$ | States space |
| $\pi$ | Policy function |
| $\mathbf{s}$ | Search step ($\mathbf{s}_k = w_{k+1} - w_k$) |
| $\tilde{r}$ | Intrinsic reward |
| $\mathcal{W}$ | Parameter of the meta-controller value function |
| $\mathbf{y}$ | Differnce between consequtive gradients ($\mathbf{y}_k = \nabla\mathcal{L}(w_{k+1}) - \nabla\mathcal{L}(w_k)$) |
| $a$ | Agent's current selected action |
| $A_t$ | A random variable for the agent's action at time $t$ |
| $a_t$ | Agent's action at time $t$ |
| $B$ | Quasi-newton matrix |
| $e$ | Agent's experience tuple, $e = (s, a, s', r')$, AKA agent's trajectory |
| $G$ | Return, accumulated future rewards |

| | |
|---|---|
| $g$ | Gradient (in chapters 5, and 6) |
| $g$ | Subgoal (in chapter 4) |
| $H$ | Hessian matrix |
| $J$ | A minibatch of data |
| $p_k$ | Search direction in $k$th iteration |
| $Q$ | State-action value function |
| $q$ | State-action value function |
| $Q(s, a; w)$ | A parameterized State-action value function |
| $Q_\pi$ | State-action value function, following policy $\pi$ |
| $q_\pi$ | State-action value function, following policy $\pi$ |
| $r$ | Reward at next time step |
| $r(s, a)$ | Reward function |
| $R_t$ | A random variable for the environment's reward at time $t$ |
| $s$ | Agent's state at current time |
| $s'$ | Agent's state at next time step |
| $S_t$ | A random variable for the agent's state at time $t$ |
| $s_t$ | Agent's state at time $t$ |
| $T$ | Final time step |
| $w$ | Parameter of the objective function, variable of the optimization problem |

# List of Figures

# List of Tables

# List of Algorithms

# Preface

The main contribution of this dissertation is introducing different methods for learning representations in the model-free Reinforcement Learning (RL) framework, and providing scalable numerical methods for solving large-scale RL tasks.

This dissertation is written to be accessible to researchers familiar with machine learning and Markov Decision Processes (MDP). However, a brief introduction to RL, MDP, and effective algorithms for *learning* is provided in chapter 2. Most chapters of this dissertation are self-contained. Methods for learning representations in chapters 3 and 4 are inspired by the computational cognitive neuroscience models of the brain. Chapter 3 investigates methods for learning sparse conjunctive representations in RL inspired by the *lateral inhibition* in the cortex. Chapter 4 studies methods for learning representations in the model-free hierarchical reinforcement learning framework, inspired by the hierarchical organization of behavior in the prefrontal cortex, and *intrinsic motivation learning*. Chapter 5 provides methods based on quasi-Newton optimization and trust-region strategy to solve empirical risk minimization problems. These optimization problems arise in many machine learning applications (including RL) and also other engineering disciplines. In this chapter, applications on image classification in the deep learning framework are considered. Chapter 6 introduces efficient numerical optimization methods in RL, and their specific implementation in deep reinforcement learning framework. Chapter 7 provides a summary of contributions from this dissertation.

We used games, such as navigation in a gridworld, or playing ATARI games, for the numerical experiments throughout the dissertation. Games are attractive environments for testing the efficiency of the AI and RL methods. But it is important to note that solving games is not a main contribution in this dissertation. The proposed methods are general and can be applied to various large-scale tasks including autonomous driving, games, and robotics.

The code for numerical simulations is available at `http://rafati.net`. Please feel free to send your questions or feedback to my email at `yrafati@gmail.com`.

If you use simulation code, or published materials, please cite the corresponding paper. (See my Publications in the Curriculum Vita.) If you use any unpublished materials, please cite the dissertation using the following citation:

- Rafati, Jacob. (2019). Learning Representations in Reinforcement Learning. Ph.D. Dissertation. University of California, Merced.

# Acknowledgment

Over the past six years I have received support and encouragement from a great number of individuals at the University of California, Merced.

Dr. David C. Noelle, my Ph.D. advisor, has been a mentor, colleague, and friend. I would like to thank him for his unwavering support and encouragement over the years. His guidance has made this a thoughtful and rewarding journey. He showed me the beautiful world of artificial intelligence, computational cognitive neuroscience, and reinforcement learning. He has introduced the open problems of the RL field and helped me to gain knowledge and courage in pursuing the most exciting research. He has supported me with my next career goals.

I would like to thank Dr. Roummel F. Marcia for all his amazing support. I have enjoyed every moment of our collaboration. I have learned a huge deal of large-scale optimization methods and the open problems of the field from him. He has encouraged me to pursue my curiosity and supported me in all means. I would like to thank Dr. Jeffrey Yoshimi, for his intellectual support. He has helped me to understand the philosophical importance of my research work. I would like to thank Dr. Marcelo Kallmann and Dr. Shawn Newsam for their great support.

I am thankful for helpful feedback from the members of the computational cognitive neuroscience laboratory. I would like to thank the staff of the School of Engineering, Tomiko Hale, Tamika Hankston, the staff of the Graduate Division, the Graduate Dean, Dr. Marjorie Zatz, and the Associate Dean, Dr. Chris Kello.

I acknowledge that the Graduate Dean's Dissertation Fellowship has supported the preparation of this dissertation. Some of my research collaboration with Dr. Roummel F. Marcia is supported by NSF Grants CMMI 1333326, and IIS 1741490. I used MERCED clusters for some of the numerical computations, supported by the NSF Grant No. ACI-1429783. I also acknowledge generous conference travel funds, and the EECS Bobcat Fellowships from the School of Engineering.

I would like to thank Omar DeGuchy for presenting our paper in EUSIPCO 2018, Dr. Johannes Brust for helpful feedback, and Dr. Harish Bhat, whose course on the mathematics of deep learning helped facilitate my research. I would like to thank Sarvani Chadalapaka for IT supports. Finally, I would like to thank Katie Williams for proofreading my manuscripts.

# Curriculum Vita

## Education

- **University of California, Merced**. Merced, CA, USA. (2013 – 2019).
  Ph.D. in Electrical Engineering and Computer Sciences.

- **Sharif University of Technology**. Tehran, Iran. (2008 – 2010).
  M.Sc. in Mechanical Engineering.

- **Sharif University of Technology**. Tehran, Iran. (2003 – 2007).
  B.Sc. in Mechanical Engineering.

## Publications

### Publications from Ph.D. Dissertation

14. **Jacob Rafati**, and David C. Noelle. (2019). Unsupervised Subgoal Discovery Method for Learning Hierarchical Representations. *7th International Conference on Learning Representations, ICLR 2019 Workshop on "Structure Priors in Reinforcement Learning", New Orleans, Louisiana.*

13. **Jacob Rafati**, and David C. Noelle. (2019). Learning Representations in Model-Free Hierarchical Reinforcement Learning. *33rd AAAI Conference on Artificial Intelligence, Honolulu, HI.*

12. **Jacob Rafati**, and David C. Noelle. (2019). Unsupervised Methods For Subgoal Discovery During Intrinsic Motivation in Model-Free Hierarchical Reinforcement Learning. *AAAI (2019) workshop on Knowledge Extraction From Games.*

11. **Jacob Rafati**, and Roummel F. Marcia. (2019). Deep Reinforcement Learning via L-BFGS Optimization. Preprint at `https://arxiv.org/abs/1811.02693`.

10. **Jacob Rafati**, and David C. Noelle (2019). Learning Representations in Model-Free Hierarchical Reinforcement Learning. Preprint at `https://arxiv.org/abs/1810.10096`.

9. **Jacob Rafati**, and Roummel F. Marcia. (2018). Improving L-BFGS Initialization For Trust-Region Methods In Deep Learning. *17th IEEE International Conference on Machine Learning and Applications, Orlando, FL.*

8. **Jacob Rafati**, Omar DeGuchy, and Roummel F. Marcia (2018). Trust-Region Minimization Algorithms for Training Responses (TRMinATR): The Rise of Machine Learning Techniques. *26th European Signal Processing Conference (EUSIPCO 2018), Rome, Italy.*

7. **Jacob Rafati**, and David C. Noelle. (2017). Sparse Coding of Learned State Representations in Reinforcement Learning. *1st Cognitive Computational Neuroscience Conference, NYC, NY.*

6. **Jacob Rafati**, and David C. Noelle. (2015). Lateral Inhibition Overcomes Limits of Temporal Difference Learning, *37th Annual Meeting of Cognitive Science Society, Pasadena, CA.*

## Publications from M.Sc. Thesis

5. **Jacob Rafati**, Mohsen Asghari and Sachin Goyal. (2014) Effects of DNA Encapsulation on Buckling Instability of Carbon Nanotube based on Nonlocal Elasticity Theory. *Proceedings of the ASME 2014 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, Buffalo, NY, USA.*

4. Mohsen Asghari, **Jacob Rafati**, and Reza Naghdabadi. (2013). Torsional Instability of Carbon Nano-Peapods based on the Nonlocal Elastic Shell Theory. *Physica E: Low-dimensional Systems and Nanostructures*, 47: p. 316-323.

3. Mohsen Asghari, Reza Naghdabadi, and **Jacob Rafati**. (2011). Small Scale Effects on the Stability of Carbon Nano-Peapods under Radial Pressure, *Physica E: Low-dimensional Systems and Nanostructures*, 43(5): pp. 1050-1055.

2. Mohsen Asghari, and **Jacob Rafati**. (2010). Variational Principles for the Stability Analysis of Multi-Walled Carbon Nanotubes Based on a Nonlocal Elastic Shell Model, *ASME 2010 10th Biennial Conference on Engineering Systems Design and Analysis (ESDA2010).*

1. **Jacob Rafati**. (2010). Stability Analysis of hybrid nanotubes based on the nonlocal continuum theories. M.Sc. Thesis. Sharif University of Technology.

# Abstract

**Learning Representations in Reinforcement Learning**
A Ph.D. dissertation by: **Jacob Rafati**
Electrical Engineering and Computer Science
University of California, Merced. 2019.
Ph.D. advisor: **Professor David C. Noelle**

Reinforcement Learning (RL) algorithms allow artificial agents to improve their action selection policy to increase rewarding experiences in their environments. Temporal Difference (TD) learning algorithm, a model-free RL method, attempts to find an optimal policy through learning the values of agent's actions at any state by computing the expected future rewards without having access to a model of the environment. TD algorithms have been very successful on a broad range of control tasks, but learning can become intractably slow as the state space grows. This has motivated methods for using parameterized function approximation for the value function and developing methods for learning internal representations of the agent's state, to effectively reduce the size of state space and restructure state representations in order to support generalization. This dissertation investigates biologically inspired techniques for learning useful state representations in RL, as well as optimization methods for improving learning. There are three parts to this investigation. First, failures of deep RL algorithms to solve some relatively simple control problems are explored. Taking inspiration from the sparse codes produced by lateral inhibition in the brain, this dissertation offers a method for learning sparse state representations. Second, the challenges of RL in efficient exploration of environments with sparse delayed reward feedback, as well as the scalability issues in large-scale applications are addressed. The hierarchical structure of motor control in the brain prompts the consideration of approaches to learning action selection policies at multiple levels of temporal abstraction. That is learning to select subgoals separately from action selection policies that achieve those subgoals. This dissertation offers a novel model-free Hierarchical Reinforcement Learning framework, including approaches to automatic subgoal discovery based on unsupervised learning over memories of past experiences. Third, more complex optimization methods than those typically used in deep learning, and deep RL are explored, focusing on improving learning while avoiding the need to fine tune many hyperparameters. This dissertation offers limited-memory quasi-Newton optimization methods to efficiently solve highly nonlinear and nonconvex optimization problems for deep learning and deep RL applications. Together, these three contributions provide a foundation for scaling RL to more complex control problems through the learning of improved internal representations.

# Chapter 1

# Introduction

Humans and non-human animals are capable of learning highly complex skills by reinforcing appropriate behaviors with reward. The midbrain dopamine system has long been implicated in reward-based learning (Schultz et al., 1993), and the information processing role of dopamine in learning has been well described by a class of reinforcement learning algorithms called *Temporal Difference (TD) Learning* (Montague et al., 1996; Schultz et al., 1997). While TD Learning, by itself, certainly does not explain all observed reinforcement learning phenomena, increasing evidence suggests that it is key to the brain's adaptive nature (Dayan and Niv, 2008).

Reinforcement learning (RL) — a class of machine learning problems — is learning how to map situations to actions so as to maximize numerical reward signals received during the experiences that an artificial agent has as it interacts with its environment (Sutton and Barto, 2017). An RL agent must be able to sense the state of its environment and must be able to take actions that affect the state. The agent may also be seen as having a goal (or goals) related to the state of the environment.

## 1.1 Motivation

One of the challenges that arise in reinforcement learning in real-world problems is that the state space can be very large. This is a version of what has classically been called the *curse of dimensionality*. Non-linear function approximators coupled with reinforcement learning have made it possible to learn abstractions over high dimensional state spaces. Formally, this function approximator is a parameterized equation that maps from state to value, where the parameters can be constructively optimized based on the experiences of the agent. One common function approximator is an artificial neural network, with the parameters being the connection weights in the network. Successful examples of using neural networks for reinforcement learning include learning how to play the game of Backgammon at the Grand Master level (Tesauro, 1995). Also, recently, researchers at DeepMind Technologies used deep convolutional neural networks (CNNs) to learn how to play some ATARI games from raw video data (Mnih et al., 2013, 2015). The resulting performance on the

games was frequently at or better than the human expert level. In another effort, DeepMind used deep CNNs and a Monte Carlo Tree Search algorithm that combines supervised learning and reinforcement learning to learn how to play the game of Go at a super-human level (Silver et al., 2016).

Despite these strengths, a mystery remains. There are some relatively simple problems for which reinforcement learning coupled with a neural network function approximator has been shown to fail (Boyan and Moore, 1995). Surprisingly, some tasks that superficially appear very simple cannot be perfectly mastered using this method. For example, learning to navigate to a goal location in a simple two-dimensional space in which there are obstacles has been shown to pose a substantial challenge to RL using a backpropagation neural network (Boyan and Moore, 1995). Using a function approximator to learn the value function has a benefit of generalization by including a bias toward mapping similar sensory states to similar predictions of future reward. At the same time, this generalization over similar states can produce a form of catastrophic interference, since there are cases in which similar states have widely different values (pattern separation). This problem is exacerbated by the fact that the values of states change as the action selection policy changes, producing a kind of destabilizing nonstationarity to the learning process.

Another major challenge in reinforcement learning is the trade-off between exploration and exploitation in an environment with sparse feedback. In order to maximize rewards, an RL agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best.

Learning to operate over different levels of *temporal abstraction* is a key challenge in tasks involving long-range planning. Hierarchical Reinforcement Learning (HRL) is an umbrella term for reinforcement learning methods that make use of some form of temporal abstraction. The acquisition of hierarchies of reusable skills is one of the distinguishing characteristics of biological intelligence (Botvinick et al., 2009), and the learning of such hierarchies is an important open problem in computational reinforcement learning. In humans, these skills are learned during a substantial developmental period in which individuals are intrinsically motivated to explore their environment and learn about the effects of their actions (Vigorito and Barto, 2010). In the context of reinforcement learning, Sutton et al. (1999) proposed the options framework, which involves abstractions over the space of actions. A suitable set of skills can help improve an agent's efficiency in learning to solve difficult problems. If an agent can develop such skill sets automatically, it should be able to efficiently solve a variety of problems without relying on hand-coded skills tailored to specific problems.

A number of methods have been suggested towards this end. One approach is to

search for commonly occurring subpolicies (or "options") in solutions to a set of tasks and to generate skills with corresponding policies (Sutton et al., 1999). A second approach is to identify subgoal states that are useful to reach and to learn skills that take the agent efficiently to these subgoals (Simsek et al., 2005; Goel and Huber, 2003).

One major open problem in hierarchical reinforcement learning is automatic option or subgoal discovery. Goel and Huber discuss a framework for subgoal discovery using the structural aspects of a learned policy model (Goel and Huber, 2003). The existing methods in the literature rely on either counting the visited states (Goel and Huber, 2003) or require construction of the graph of the state transitions (Simsek et al., 2005; Machado et al., 2017). These methods have been successful in finding doorways that act as good subgoals when navigating between rooms, but they do not have the flexibility needed to find a broad range of other kinds of subgoals. The lack of generalization and expensive memory requirements, as well as the lack of sensitivity to the similarity and novelty across experiences, causes these methods to fail to discover good subgoals in large-scale reinforcement problems.

Reinforcement learning is different from the most commonly used machine learning paradigm: supervised learning. Supervised learning is learning from a training set of labeled examples provided by a knowledgeable external supervisor. In contrast, in the reinforcement learning problem it is generally taken as impractical to obtain supervised information about desired behavior that is both correct and representative of all the situations (states) in which the agent is expected to act (Sutton and Barto, 2017). Indeed, learning this optimal behavior is the goal of reinforcement learning.

Learning representations through the use of large-scale optimization methods, as done in deep learning and reinforcement learning, is typically restricted to the class of first-order algorithms, including the popular stochastic gradient decent method. Although they are preferred for their ease of implementation and their relatively low computational costs, they are not without their challenges. These types of methods are heavily dependent on the tuning and initialization of network learning parameters. These methods rely on gradient information because of the computational complexity associated with calculating the second derivative Hessian matrix inversion, as well as the memory storage required in large-scale problems. The benefit of using second derivative information is that these methods can result in improved convergence properties for problems typically found in a non-convex setting such as saddle points and local minima. As an alternative to gradient descent, limited memory quasi-Newton algorithms with *line search* or *trust-region* have been implemented in a deep learning setting (Rafati et al., 2018) and applied to classification tasks. These methods approximate second derivative information, improving the quality of each training iteration and circumventing the need for application-specific parameter tuning. Given that the quasi-Newton methods are efficient in supervised learning problems, an important question arises: Is it also possible to use the quasi-Newton methods to learn the state representations in reinforcement learning successfully? We investigated this question as part of this dissertation research project.

## 1.2 Dissertation Outline, and Objectives

Chapter 2 provides background knowledge on reinforcement learning problems. The first part of this chapter introduces the reinforcement learning problem within the Markov Decision Processes (MDP) framework. Then the temporal difference learning algorithms for solving the model-free reinforcement learning problem is introduced. Some brief background information on artificial neural networks and convolutional neural networks is provided.

In Chapter 3, I demonstrate how incorporating a ubiquitous feature of biological neural networks into the artificial neural networks used to approximate the value function can allow reinforcement learning to succeed at simple tasks that have previously challenged it. Specifically, I show that the incorporation of *lateral inhibition*, producing competition between neurons so as to produce *sparse conjunctive representations*, can produce success in learning to approximate the value function using an artificial neural network, where only failure had been previously found. Through computational simulation I provide preliminary evidence that learning a sparse representation of states through a mechanism inspired by lateral inhibition in the brain may help compensate for a weakness of using neural networks in TD Learning.

In Chapter 4, I first provide background information concerning the hierarchical reinforcement learning problem. Inspired by Kulkarni et al. (2016), I introduce the meta-controller/controller framework, a two-level HRL technique for temporal abstraction. Then I propose a hierarchical reinforcement learning framework that makes it possible to simultaneously perform subgoal discovery, learn intrinsic motivation, and succeed at meta-policy learning. I offer an original approach to HRL that does not require the acquisition of a model of the environment, suitable for large-scale applications. I demonstrate the efficiency of our method on two RL problems with sparse delayed feedback: a variant of the rooms environment and the first screen of the ATARI 2600 *Montezuma's Revenge* game. Our proposed method of automatic subgoal discovery uses only the limited memory of past experiences of the learner. The objective is to provide a scalable subgoal discovery method using unsupervised learning and novelty detection methods that are suitable for large-scale reinforcement learning problems. I briefly discuss the neural correlates of the model-free computational HRL framework. I show how intrinsic motivation learning produces an efficient policy for exploring the environment, and I present an approach to learning the structure of the state space using unsupervised subgoal discovery.

In Chapter 5, I introduce novel large-scale limited-memory BFGS (L-BFGS) optimization methods using the trust-region and the line-search strategies – as an alternative to the gradient descent methods. I implement practical computational algorithms to solve the Empirical Risk Minimization problems that arise in machine learning and deep learning applications. I provide empirical results on the classification task of the MNIST dataset and show robust convergence with preferred generalization characteristics. Based on the empirical results, I provide a comparison between the trust-region strategy with the line-search strategy on their different convergence properties. I also

study techniques for initialization of the positive definite L-BFGS quasi-Newton matrices in the trust-region strategy so that they do not introduce any false curvature conditions when constructing the quadratic model of the objective function.

In Chapter 6, I introduce novel quasi-Newton optimization methods for deep reinforcement learning in the line-search framework. Our method bridges the disparity between first order methods and second order methods by continuing to use gradient information to calculate low-rank Hessian approximations. I implement practical computational algorithms to solve the Empirical Risk Minimization problems in deep RL. The objective is learning the representations of the value function without the need to fine tune the learning hyper-parameters. I offer methods for learning representations that do not need experience replay mechanism in order to reduce the computation time and the experience memory size. I provide a formal convergence analysis, as well as empirical results on a subset of the classic ATARI 2600 games.

In Chapter 7, I provide concluding remarks and a summary of the contributions of this dissertation.

# Chapter 2

# Reinforcement Learning

In this part, I provide a formal description of reinforcement learning. Most of the formalizations are taken from Sutton and Barto's *Introduction to Reinforcement Learning* book (Sutton and Barto, 2017).

## 2.1 Reinforcement Learning Problem

### 2.1.1 Agent and Environment interaction

The reinforcement learning problem is that of learning through interaction with an *environment* how to achieve a *goal*. The learner and decision maker is called the *agent* and everything outside of the agent is called the *environment*. The agent and environment interact over a sequence of discrete time steps, $t = 0, 1, 2, \ldots, T$.

At each time step, $t$, the agent receives a representation of the environment's *state*, $S_t = s$ from the set of all possible states, $\mathcal{S}$, and on that basis the agent selects an *action*, $A_t = a$, from the set of all available actions for the agent, $\mathcal{A}$. One time step later, at $t + 1$, as a consequence of the agent's action, the agent receives a *reward*, $R_{t+1} = r \in \mathbb{R}$, and also an update on the agent's new state, $S_{t+1} = s'$, from the environment. Each cycle of interaction is called an *experience* (or a trajectory), $e = \{s, a, r, s'\}$. Figure 2.1 summarizes the agent/environment interaction. It is useful to think of the state as equivalent to the *state of a dynamical system*, but, here, we refer to "state" as whatever information is available to the agent, beyond reward.

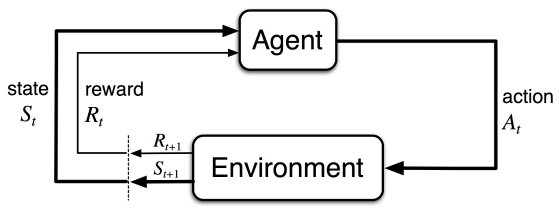



Figure 2.1: The agent/environment interaction in reinforcement learning. Adopted from (Sutton and Barto, 1998)

### 2.1.2 Policy Function

At each time step, the agent uses a mapping from states to possible actions, $\pi_t : \mathcal{S} \to \mathcal{A}$. This mapping is called the agent's *policy*. The stochastic policy can be defined as the probability of taking action $A_t = a$ at state $S_t = s$

$$\pi_t(a|s) = p(A_t = a|S_t = s). \tag{2.1}$$

A deterministic policy can be obtained as

$$a = \pi_t(s) = \arg\max_{a'} \ \pi_t(a'|s) \tag{2.2}$$

### 2.1.3 Objective in RL

The objective of the reinforcement learning problem is the maximization of the expected value of *return*, i.e. the cumulative sum of the received reward

$$G_t = \sum_{t'=t+1}^{T} \gamma^{t'-t-1} R_{t'}, \tag{2.3}$$

where $T \in \mathbb{N}$ is a final step and $\gamma \in (0, 1]$ is a discount factor. As $\gamma \to 1$, the objective takes future rewards into account more strongly (farsighted agent) and if $\gamma \to 0$ the agent is only concerned about maximizing the immediate reward (myopic agent). Having $\gamma < 1$ can bound the return $G_t < \infty$ when $T \to \infty$. The goal of the reinforcement learning problem is finding an optimal policy, $\pi^*$ that maximizes the expected return

$$\pi^* = \arg\max_{\pi} \mathbb{E}_\pi[G_t]. \tag{2.4}$$

## 2.2 Markov Decision Processes

In this section, I introduce the formal problem of finite Markov decision processes (MDP). MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. I introduce key elements of the problem's mathematical structure, such as returns, value functions, and Bellman equations. I also introduce the value iteration algorithm for solving MDPs in tabular fashion.

### 2.2.1 Formal Definition of MDP

A finite Markov decision process is a 5-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $P = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the

transition probability that action $a$ in state $s$ at time $t$ will lead to state $s'$ at time $t+1$, $R(s,a)$ is the immediate reward received after transitioning from state $s$ to state $s'$, due to action $a$, $\gamma \in (0,1]$ is the discount factor, which represents the difference in importance between future rewards and present rewards.

A state signal that succeeds in retaining all relevant information from the past is said to be Markov, or to have the Markov property. Consider how a general environment might respond at time $t+1$ to the action taken at time $t$. In general, the dynamics can be defined only by specifying the complete joint probability distribution $\Pr(S_{t+1} = s', R_{t+1} = r \mid S_{0:t}, A_{0:t})$, but if the state signal has the Markov property, then the environment's response at $t+1$ depends only on the state and action representations at $t$, in which case the environment's dynamics can be defined by:

$$p(s', r|s, a) = \Pr(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a) \quad \forall s', r. \tag{2.5}$$

If the state and action spaces are finite, then it is called a finite Markov decision process. In this dissertation, I implicitly assume that the environment is a finite MDP. In the case of continuous environments, I assume that the discretized environment is a finite MDP. The probabilities given by the four-argument function $p$ completely characterize the dynamics of a finite MDP, such as the *state-transition probabilities* $p(s'|s,a)$

$$p(s'|s, a) = \sum_r p(s', r|s, a). \tag{2.6}$$

We can also compute the expected rewards for state-action pairs as a two-argument function $r(s,a)$

$$r(s, a) \triangleq \mathbb{E}[R_{t+1} = r|S_t = s, A_t = a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a), \tag{2.7}$$

or the expected reward for state-action-next-state triples, $r(s,a,s')$

$$r(s, a, s') \triangleq \mathbb{E}[R_{t+1} = r|S_t = s, A_t = a, S_{t+1} = s'] = \sum_{r\in\mathcal{R}} r\frac{p(s', r|s, a)}{p(s'|s, a)}. \tag{2.8}$$

### 2.2.2 Value Function.

The value of state $s$ under a policy $\pi$, denoted $v_\pi(s)$ is the expected return when starting from $s$ and following $\pi$ thereafter. For MDPs, $v_\pi(s)$ can be defined as

$$\begin{aligned} &v_\pi : \mathcal{S} \to \mathbb{R} \\ &v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{t'=t+1}^{T} \gamma^{t'-t-1} R_{t'} \;\middle|\; S_t = s\right], \end{aligned} \tag{2.9}$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy $\pi$. The value of a terminal state is, by definition, always zero. The function $v_\pi$ is called the state-value function for policy $\pi$. Similarly, we can define the state-action value function $q_\pi(s, a)$ for policy $\pi$ as

$$\begin{aligned} &q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R} \\ &q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi\Bigg[\sum_{t'=t+1}^{T} \gamma^{t'-t-1} R_{t'} \;\Bigg|\; S_t = s, A_t = a\Bigg], \end{aligned} \tag{2.10}$$

Based on these definitions, the two value functions, $v_\pi$ and $q_\pi$ are related

$$v_\pi(s) = \sum_a \pi(a|s)\; q_\pi(s,a). \tag{2.11}$$

Note that, in the deterministic case,

$$v_\pi(s) = \max_a q_\pi(s,a). \tag{2.12}$$

### 2.2.3 Bellman Equations

For any policy $\pi$ and any state $s$, the following consistency condition holds between the value of $s$ and the value of its possible successor states $s'$:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \quad \forall s \in \mathcal{S} \tag{2.13}$$

$$q_\pi(s,a) = \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \sum_{a'} \pi(a'|s')\; q_\pi(s', a')\Big], \quad \forall (s,a) \in \mathcal{S} \times \mathcal{A}. \tag{2.14}$$

Note that (2.11) holds for Bellman's equations and we can rewrite the Bellman equations (2.13) and (2.14) using the expected reward and state transition probabilities as

$$v_\pi(s) = \sum_a \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'} p(s'|s,a) v_\pi(s')\Big], \quad \forall s. \tag{2.15}$$

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} q_\pi(s', a'), \quad \forall s, a. \tag{2.16}$$

Note that, when the policy is deterministic, Bellman's equations can be written as

$$v_\pi(s) = \Big[r(s,a) + \gamma \sum_{s'} p(s'|s,a) v_\pi(s')\Big], \quad \forall s. \tag{2.17}$$

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} q_\pi(s', a'), \quad \forall s, a. \tag{2.18}$$

### 2.2.4 Optimal Value Function

The optimal state-value function, denoted $v^*$ is the objective function in a reinforcement learning problem

$$v^* = \max_\pi v_\pi(s), \quad \forall s. \tag{2.19}$$

There is always at least one policy that is better than or equal to all other policies. This is an optimal policy

$$\pi^* = \arg\max_\pi v_\pi(s), \quad \forall s. \tag{2.20}$$

Optimal policies also share the same optimal action-value function, denoted as $q^*$

$$q^* = \max_\pi q_\pi(s, a), \quad \forall s, a. \tag{2.21}$$

One can easily obtain optimal policies once the optimal value functions, $v^*$ or $q^*$ are found. These functions satisfy the Bellman optimality equations

$$v^*(s) = \max_a \left\{ r(s, a) + \gamma \sum_{s'} p(s'|s, a) v^*(s') \right\}, \quad \forall s. \tag{2.22}$$

$$q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|s, a) \max_{a'} q^*(s', a'), \quad \forall s, a. \tag{2.23}$$

### 2.2.5 Value iteration algorithm

To compute $v^*$, one can choose an initial guess $v^0(s)$ and then iteratively apply

$$v^{i+1}(s) = \mathbb{T}[v^i](s), \tag{2.24}$$

where the operator $\mathbb{T}$ is defined by

$$\mathbb{T}[v](s) = \max_a \left\{ r(s, a) + \gamma \sum_{s'} p(s'|s, a) v(s') \right\}, \quad \forall s. \tag{2.25}$$

To be clear, $\mathbb{T}$ is an operator that takes a function $v : \mathcal{S} \to \mathbb{R}$ and produces a function $\mathbb{T}[v] : \mathcal{S} \to \mathbb{R}$ as output. It's easy to prove that, if $0 < \gamma < 1$, the operator $\mathbb{T}[v]$ is a contraction. Hence, $\mathbb{T}$ has a unique fixed point which is the same as the optimal value function (based on the definition of the optimal value function in (2.22)). The contraction mapping principle guarantees that

$$\lim_{i \to \infty} v^i(s) = v^*(s), \tag{2.26}$$

i.e., that the value iteration given by (2.24) converges to the optimal value function. Once $v^*$ is known, we can compute $q^*(s, a)$ (defined in (2.23)) as

$$q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|s, a) \max_{a'} q^*(s', a'), \tag{2.27}$$

where $q^*$ is the state-action optimal value function, also known as optimal $Q$-function. Note that, we can compute $v^*$ if $q^*$ is known

$$v^*(s) = \max_a q^*(s, a). \tag{2.28}$$

In order to compute $q^*$ directly, one can choose an initial guess $q^0(s, a)$ and then iteratively apply

$$q^{i+1}(s, a) = \mathbb{H}[q^i](s, a), \tag{2.29}$$

where the operator $\mathbb{H}$ is defined by

$$\mathbb{H}[q](s, a) = r(s, a) + \gamma \sum_{s'} p(s'|s, a) \max_{a'} \{q(s', a')\}, \quad \forall s, a. \tag{2.30}$$

The operator $\mathbb{H}$ is also a contraction if $0 < \gamma < 1$ and has a unique fixed point which is same as the optimal $Q$-function, i.e. $q^*$ defined in (2.23). Once the optimal value function $v^*$ or $q^*$ are computed, one can also find the optimal policy map function $\pi_*$:

$$\pi_*(s) = \arg\max_a \left\{ r(s, a) + \gamma \sum_{s'} p(s'|s, a) v(s') \right\}, \quad \forall s. \tag{2.31}$$

$$\pi_*(s) = \arg\max_a q_*(s, a), \quad \forall s. \tag{2.32}$$

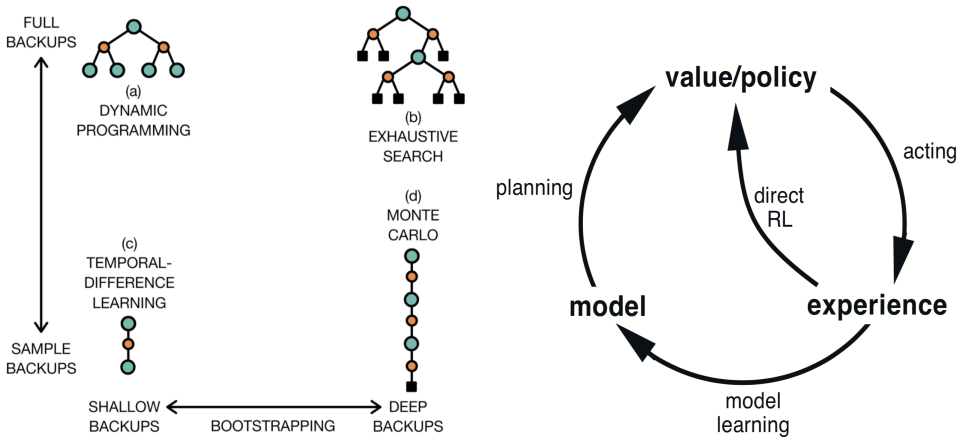
## 2.3 Reinforcement Learning algorithms

Reinforcement learning involves learning to map situations to actions in order to maximize a numerical reward signal (see Figure 2.1). Reinforcement Learning is a class of solution methods for solving MDPs, when the agent does not have prior access to the environment models, i.e. the state transition probabilities, $p(s'|s, a)$, and the reward function, $r(s, a)$. The agent usually does not have access to the state space $\mathcal{S}$.

Instead, the agent only perceives experiences (or trajectories) from interaction with the environment. The agent can save a limited memory of past experiences (or history) in the set $\mathcal{D}$. It is important to note that each experience, $(s, a, s', r)$, is an example of the probability distribution $p(s', r|s, a)$. Thus the experience memory plays the role of the training data in RL.

These limitations make learning in RL more challenging than in MDP. The agent should take actions to gain more rewards. At the same time it should take actions in order to explore the state space. This is called the exploration-exploitation trade off problem. Another major issue is when the state space is huge, and storing the state-action values in a table is not possible.

RL methods can be categorized with regards to (1) their depth of search (shallow search or deep search), and (2) whether they use sample or full backup. Figure 2.2 shows classes of methods along these two dimensions. If the environment, or model of it, i.e. the state transition probabilities, $p(s'|s, a)$, and reward function $r(s, a, s')$




(a)–(d) (e)

Figure 2.2: Two dimensions of RL algorithms. At the extremes of these dimensions are (a) dynamic programming, (b) exhaustive search, (c) one-step TD learning and (d) pure Monte Carlo approaches. Adopted from (Arulkumaran et al., 2017). (e) Relationships among direct learning (model-free methods), and planning (model-based methods). Adopted from (Sutton and Barto, 2017).

is known to the agent, one can find the optimal policy by a brute search (exhaustive search) (Figure 2.2 (b)). The exhaustive search gets exponentially extensive for larger state spaces. There are methods based on Dynamical Programming (DP) that take into consideration the structural properties of the value function, and search for optimal policies more efficiently (Bertsekas, 2000) (Figure 2.2 (a)). The curse of dimensionality, and the fact that agent might not have access to environment models, motivates methods based on sampling, and shallow DP.

In the remainder of this section, I introduce some of the most common RL methods, describing their approaches to solving MDP problems without access to environment models.

### 2.3.1 Value-based vs Policy-based Methods

There are two main approaches to solving RL problems: methods based on value functions and methods based on policy search. There is also a hybrid, actor-critic approach, which employs both value functions and policy search. Policy search algorithms, such as *policy gradient* (Schulman et al., 2015) do not need to maintain a value function model, but directly search for an optimal policy.

In this dissertation, I focus on value-based methods, due to their popularity, efficiency, and interesting convergence properties.

### 2.3.2 Bootstrapping vs Sampling

Algorithms based on Dynamic Programming can exploit the properties of the optimal value function in the Bellman equation in Markovian environments. This leads to methods that can learn the value function by bootstrapping. Temporal Difference learning is an example of this class, which will be introduced in Section 2.4.

Instead of bootstrapping the value function using dynamic programming methods, one can use Monte Carlo methods to estimate the expected return from a state by averaging the return from multiple rollouts of a policy. Because of this, pure Monte Carlo methods can also be applied in non-Markovian environments (Arulkumaran et al., 2017). It is possible to get the best of both methods by combining TD learning and Monte Carlo policy evaluation, as done in the TD($\lambda$) algorithm. See (Sutton and Barto, 2017) for more details.

### 2.3.3 Model-Free vs Model-Based RL

If a good model of the environment is given, it is possible to use dynamic programming over all possible actions, sample trajectories for heuristic search (as was done by AlphaGo (Silver et al., 2016)), or even perform an exhaustive search (Figure 2.2 (b)).

In RL, these models are not given. But, the interactions with the environment can be used to learn a model of the environment. In *model-based* RL methods, the agent attempts to learn a model of the environment, in order to simulate transitions, using the learned model, and increase sample efficiency. In *model-free* RL methods, the agent learns directly from the interactions with the environment, and does not learn a model. Model-based methods rely on *planning*, while model-free methods primarily rely on *learning* (Sutton and Barto, 2017). See Figure 2.2 (e) for the information flow in model-free and model-based learning.

Model-based RL is particularly important in domains where interaction with the environment is expensive. For example, in autonomous driving applications, it is safer to plan using the learned model before executing a potentially dangerous action and observing the consequence of that action from the environment. However, learning a model introduces extra complexities, and an erroneous model might have undesired effects on the learned policy (or value).

The agent in model-based RL does not need to learn a value function. The expectation of the future rewards can be computed by simulating the model to produce simulated experiences. However, there are methods such as Dyna, which integrates model-based (planning) and model-free (learning) methods, allowing the agent to learn the value function by both interacting with the environment and simulating the learned model. One can use the methods based on Dynamic Programing (bootstrapping) to learn these value function. See (Sutton, 1991; Sutton and Barto, 2017) for more details.

In this dissertation, I am mainly interested in model-free RL methods (direct learning), specifically Temporal Difference learning. TD algorithms are among the most prominent RL methods, since they are simple, efficient, powerful, and provide

an account of the role of dopamine in learning in the brain. In the next section, I introduce two popular TD algorithms: Q-learning and SARSA.

## 2.4 Temporal Difference Learning

Reinforcement learning can solve Markov decision processes without explicit specification of the transition probabilities. Temporal Difference (TD) learning (Sutton, 1988) is a model-free reinforcement learning algorithm that attempts to learn a policy without learning a model of the environment. TD is a combination of Monte Carlo ideas (repeated random sampling) and dynamic programming ideas. (See Figure 2.2 (c), and (d)). The goal of the TD approach is to learn to predict the value of a given state based on what happens in the next state by bootstrapping, that is updating values based on the learned values, without waiting for a final outcome. TD methods have an advantage over DP methods in that they do not require a model of the environment, or of its reward and next-state probability distributions. The advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an on-line, fully incremental fashion. Both TD and Monte Carlo methods use experience to solve the prediction problem.

Given some experience following a policy $\pi$, both methods update their estimate $V$ of $v_\pi$ for the nonterminal states, $s_t$, occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(s_t)$. A simple every-visit Monte Carlo method suitable for non-stationary environments is

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)] \tag{2.33}$$

where $G_t$ is the actual return following time $t$, and $\alpha$ is a constant step-size parameter. In comparison, TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward $r_{t+1}$ and the estimate $V(s_{t+1})$. The simplest TD method makes the update

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.34}$$

immediately on transition to $s_{t+1}$ and receiving $r_{t+1}$. The quantity

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

is called the *TD error*. I provide two common algorithms that implements the TD procedure with one-step backup (known as TD(0)).

### 2.4.1 SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy TD algorithm that learns the action-value function. The name reflects the fact that the main function for

updating the $Q$-value depends on the current state of the agent $s$, the action the agent chooses $a$, the reward $r$ the agent gets for choosing this action, the state $s'$ that the agent will be in after taking that action, and finally the next action $a'$ the agent will choose in its new state, hence $(s, a, r, s', a')$. As in all on-policy methods, the $Q$-function is continually estimated for the behavior policy using

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]. \tag{2.35}$$

The reason that SARSA is on-policy is that it updates its $Q$-values using the $Q$-value of the next state $s'$ and the current policy's action $a'$. Algorithm 1 gives the SARSA procedure.

**Algorithm 1** SARSA: On-Policy TD Learning

Initialize $Q(s,a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
**repeat** (for each episode)
  initialize $s$
  $a \leftarrow$`EPSILON-GREEDY`$(Q(s,\mathcal{A}),\epsilon)$ (see Algorithm 3)
  **repeat** (for each step of episode)
    take action $a$, observe reward $r$ and next state $s'$
    $a' \leftarrow$`EPSILON-GREEDY`$(Q(s',\mathcal{A}),\epsilon)$
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$, $a \leftarrow a'$
  **until** ($s$ is terminal or reaching to max number of steps)
**until** (convergence or reaching to max number of episodes)

### 2.4.2 Q-Learning

$Q$-Learning is a unifying TD control algorithm which simultaneously optimizes the value function and the policy. $Q$-Learning is often used in an off-policy manner, learning about the greedy policy while the data is generated by a different policy that ensures exploration. Algorithm 2 gives the $Q$-learning procedure.

## 2.5 Generalization in Reinforcement Learning

The TD learning algorithms need to maintain an estimate of the value function, and this function could be stored as a simple look-up table. However, when the state space is large or not all states are observed during training, storing these estimated values in a table is no longer possible. One can, instead, use a function approximator to represent the mapping to the estimated value. A parameterized functional form with weight vector $w$ can be used: $v(s;w) \approx v_\pi(s)$ or $q(s,a;w) \approx q_\pi(s,a)$. In this case, TD learning requires a search through the space of parameter values, $w$, for the function approximator. One common way to approximate the value function is to use

**Algorithm 2** Q-Learning: Off-Policy TD Learning

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
**repeat** (for each episode)
  initialize $s$
  **repeat** (for each step of episode)
    $a \leftarrow$`EPSILON-GREEDY`$(Q(s, \mathcal{A}), \epsilon)$ (see Algorithm 3)
    take action $a$, observe reward $r$ and next state $s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  **until** ($s$ is terminal)
**until** (convergence or reaching to max number of episodes)

**Algorithm 3** The $\epsilon$-greedy policy function

Input $Q(s, a)$, action space $\mathcal{A}$, exploration rate $\epsilon \in [0, 1]$
**if** `rand()` $< \epsilon$ **then**
  **return** a random action $a \in \mathcal{A}$
**else**
  **return** $a \leftarrow \arg\max_a Q(s, a)$
**end if**

a nonlinear functional form, such as that embodied by an artificial neural network. When a change is made to the weights based on an experience from a particular state, the changed weights will then affect the value function estimates for similar states, producing a form of generalization. Such generalization makes the learning process potentially much more powerful.

### 2.5.1 Feed-forward Neural Networks

Feed-forward neural networks are called networks because they are typically represented by composing together many different functions (Goodfellow et al., 2016). The goal of a feed-forward network is to approximate some function $q^*$. Consider a value function $q^*(s)$ that maps the state $s$, as an input, to the corresponding $q(s, a)$ values for every possible action. A feed-forward network defines a mapping $\mathbf{Q} = [q(s, a_1; w), q(s, a_2; w), \dots]$, and it can learn the values of the parameters, $w$, that result in the best approximation of the target function. The defining characteristics of different feed-forward artificial neural network architectures are the number of hidden layers and the type of layers used (convolutional, fully connected, etc). Each layer is described by the following expression

$$h^{(i)} = \theta^{(i)}(h^{(i-1)}, w^{(i)}), \quad \text{for } i = 1, \dots, n, \tag{2.36}$$

where $n$ is number of the layers, $h^{(i-1)}$ is the input to the current layer, $h^{(i)}$ is the output of the current layer, and $w^{(i)}$ corresponds to the weight parameters of the

layer, which are adjusted during training. Note that $h^0$ is the input to the whole network (e.g., a representation of the state). The choice of layer architecture, $\theta^{(i)}$, can include fully connected layers, convolutions, pooling layers, and autoencoders. The layer architecture can also include an *activation function*, such as a rectified linear unit (ReLU), a logistic sigmoid, or the softmax function. Having incorporating indices to the layer components one can represent an artificial neural network with $n$ layers as

$$\begin{aligned} h^{(1)} &= \theta^{(1)}(h^{(0)}, W^{(1)}), \\ h^{(2)} &= \theta^{(2)}(h^{(1)}, W^{(2)}), \\ &\vdots \\ h^{(n)} &= \theta^{(n)}(h^{(n-1)}, W^{(n)}), \end{aligned} \tag{2.37}$$

During the training of the network, the information $h^{(0)} = s$ is passed through the layers to obtain the network's approximation of the desired output, $h^{(n)} = \mathbf{Q}$. The objective is to adjust the weights $w^{(i)}$ in order to improve the quality of the output, $h^{(n)}$. Learning is accomplished through an optimization procedure, minimizing a loss function, $L$.

### 2.5.2 Loss Function as Expectation of TD Error

The network can be trained by minimizing a loss functions $L(w)$ that changes at each training iteration,

$$L(w) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\frac{1}{2}\Big(y - q(s,a;w)\Big)^2\Big], \tag{2.38}$$

where

$$y = \Big[r + \gamma q(s', a'; w^-)\Big], \tag{2.39}$$

is the target values (for SARSA learning method), $w^-$ is the learning parameters from the previous iteration, and $\mathcal{D}$ is *experience replay memory*. Note that for the $Q$-Learning method, the following target values should be used

$$y = [r + \gamma \max_{a'} q(s', a'; w^-)]. \tag{2.40}$$

It is common in the deep learning literature to collect and save experiences in a replay memory $\mathcal{D}$ and update the weights of the network by replaying this memory (Mnih et al., 2013, 2015). At each training iteration, an experience $e = (s, a, r, s')$ is sampled uniformly from the replay memory $D$. The parameters from the previous iteration, $w^-$, are held fixed when optimizing the loss function, $L(w)$. Note that the targets depend on the network weights. This is in contrast with the targets used for supervised learning, which are fixed before learning begins.

## 2.6 Empirical Risk Minimization in Deep RL

In practice, instead of minimization of the expected risk in (2.38) one can define an optimization problem for the *empirical risk* as follows

$$\min_{w\in\mathbb{R}^n} L(w) \triangleq \frac{1}{2N}\sum_{e\in\mathcal{D}}\Big[\Big(y - q(s,a;w)\Big)^2\Big], \tag{2.41}$$

where $n$ is the total number of parameters, and $N = |\mathcal{D}|$ is the size of data. In order to produce a good approximation of the value function, one should solve the empirical risk minimization problem (2.41) using a TD learning algorithm. In large-scale optimization problems, $n$, and $N$ are very large numbers. Additionally, $L(w)$ is a nonconvex and highly nonlinear function. The most simple and common method used in the reinforcement learning literature is the Stochastic Gradient Descent (SGD) method. SGD requires computation of the gradient of the objective function, $\nabla L(w)$ in (2.41), with respect to the parameters, $w$. Rather than computing the full gradient, it is often computationally expedient to optimize the loss function stochastically. It is common to sample a mini-batch $J \subset \mathcal{D}$ and compute $\nabla^{(J)}L(w) \approx \nabla L(w)$, using the experiences from $J$ as follows

$$\nabla^{(J)}L(w) = \frac{-1}{|J|}\sum_{(s,a,r,s')\in J}\Big[\Big(y - q(s,a;w)\Big)\nabla q(s,a;w)\Big]. \tag{2.42}$$

**Algorithm 4** Q-Learning with Experience Replay

```
Initialize replay memory D
Initialize parameters of q(s, a; w) arbitrarily
repeat (for each episode)
    initialize s
    a ←EPSILON-GREEDY(q(s, A; w), ϵ)
    repeat (for each step of episode t = 1, . . . , T)
        take action a, observe reward r and next state s′
        store experience e = (s, a, r, s′) to experience replay memory D
        sample a random mini-batch from J ⊂ D and compute ∇_w L(w) in (2.42)
        update weights (e.g. SGD step) w ← w − α∇_w L(w)
        s ← s′
    until (s is terminal)
until (convergence or reaching to max number of episodes)
```

Algorithm 4 outlines Q-learning with experience replay. Although methods based on SGD are simple, they have undesirable convergence properties especially in large-scale problems. These methods require tuning so many hyperparamters, and they require large memory in order to replay experiences. In Chapters 5 and 6, I study alternative methods for solving the large-scale nonconvex optimization problem (2.41) to improve the performance of learning, as well as improve the convergence properties.

# Chapter 3

# Learning Sparse Representations in Reinforcement Learning


## Abstract

Reinforcement learning (RL) algorithms allow artificial agents to improve their selection of actions to increase rewarding experiences in their environments. Temporal Difference (TD) Learning – a model-free RL method – is a leading account of the midbrain dopamine system and the basal ganglia in reinforcement learning. These algorithms typically learn a mapping from the agent's current sensed state to a selected action (known as a *policy* function) via learning a value function (expected future rewards). TD Learning methods have been very successful on a broad range of control tasks, but learning can become intractably slow as the state space of the environment grows. This has motivated methods that learn internal representations of the agent's state, effectively reducing the size of the state space and restructuring state representations in order to support generalization. However, TD Learning has been shown to fail to learn some fairly simple control tasks, challenging this explanation of reward-based learning. We hypothesize that such failures do not arise in the brain because of the ubiquitous presence of lateral inhibition in the cortex, producing sparse distributed internal representations that support the learning of expected future reward. The sparse conjunctive representations can avoid catastrophic interference while still supporting generalization. We provide support for this conjecture through computational simulations, demonstrating the benefits of learned sparse representations for three problematic classic control tasks: Puddle-world, Mountain-car, and Acrobot.


## 3.1 Introduction

Humans and non-human animals are capable of learning highly complex skills by reinforcing appropriate behaviors with reward. The midbrain dopamine system has long

been implicated in reward-based learning (Schultz et al., 1993), and the information processing role of dopamine in learning has been well described by *Temporal Difference (TD) Learning* (Montague et al., 1996; Schultz et al., 1997). While TD Learning, by itself, certainly does not explain all observed reinforcement learning phenomena, increasing evidence suggests that it is key to the brain's adaptive nature (Dayan and Niv, 2008).

Beyond empirical support for the TD Learning account of biological reinforcement learning, the power of this learning method suggests that it may be capable of explaining the neural basis of the successful learning of even fairly complex tasks (Sutton and Barto, 1998). This algorithm can learn elaborate decision making skills, such as playing the game of Backgammon at the Grand Master level (Tesauro, 1995). There are even proofs that TD Learning will converge to optimal performance, given enough experience (Dayan, 1992).

Despite these strengths, a mystery remains. There are some relatively simple problems for which TD Learning has been shown to fail (Boyan and Moore, 1995). These problems arise when the space of possible sensory states of the learning agent is so large that it is intractable to store the agent's learned assessment of the *value* or *quality* of each state (i.e., its expectation of future reward, given that it is in that state) in a large look-up table. In these cases, it is necessary to encode the agent's learned *value function*, mapping from sensory state features to an expectation of future reward, using some form of function approximator. As previously reviewed, this function approximator is formally a parameterized equation that maps from state to value, where the parameters can be optimized based on the experiences of the agent. As discussed in Chapter 2, a commonly used function approximator is an artificial neural network, with the parameters being the connection weights. Such a network, adapted using a version of stochastic gradient descent called the *backpropagation of error* learning method (Rumelhart et al., 1986), was used in the previously mentioned Backgammon playing program (Tesauro, 1995). As illustrated by this program, TD Learning with an artificial neural network approximating the value function, can solve apparently complex tasks. Using a function approximator to learn the value function has the added benefit of potentially supporting generalization by including a bias toward mapping similar sensory states to similar predictions of future reward.

Surprisingly, some tasks that superficially appear very simple cannot be perfectly mastered using this method. For example, learning to navigate to a goal location in a simple two-dimensional space in which there are obstacles has been shown to pose a substantial challenge to TD Learning using a backpropagation neural network (Boyan and Moore, 1995). Note that the proofs of convergence to optimal performance depend on the agent maintaining a potentially highly discontinuous value function in the form of a large look-up table, so the use of a function approximator for the value function violates the assumptions of those formal analyses. Still, it seems unusual that this approach to learning can succeed at some difficult tasks but fail at some other fairly easy tasks.

The power of TD Learning to explain biological reinforcement learning is greatly

reduced by this observation. If TD Learning fails at simple tasks that are well within the reach of humans and non-human animals, then it cannot be used to explain how the dopamine system supports such learning.

In this chapter, we demonstrate how incorporating a ubiquitous feature of biological neural networks into the artificial neural networks used to approximate the value function can allow TD Learning to succeed at simple tasks that have previously challenged it. Specifically, we show that the incorporation of *lateral inhibition*, producing competition between neurons so as to produce *sparse conjunctive representations*, can produce success in learning to approximate the value function using an artificial neural network, where only failure had been previously found. Thus, through computational simulation, we provide preliminary evidence that lateral inhibition may help compensate for a weakness of TD Learning, improving this machine learning method and further buttressing the TD Learning account of dopamine-based reinforcement learning in the brain.

In the remainder of this chapter, we initially provide some background concerning the reported failure of TD Learning on fairly simple problems. We then provide details concerning our computational simulations of TD Learning with lateral inhibition included. The results of these simulations, comparing the performance of our approach to previously examined methods, are then described. We close this chapter with a discussion of the reported results and ideas for future work in this area.

## 3.2 Background

Consider a very simple two-dimensional "grid world" environment that remains static over time. A reinforcement learning agent in this environment may be faced with the choice, at each time step, to move a fixed distance either North, South, East, or West. On each time step, the agent receives mild negative reinforcement, until it moves to a goal location in the Northeast corner of the space, at which point the agent is relieved of negative reinforcement. Further, imagine that this environment contains "puddles" through which the agent can move, but moving into puddle locations produces extremely strong negative reinforcement. Finally, consider the case in which the agent can perfectly sense its location in the grid environment (i.e., its Cartesian coordinates), but it otherwise has no senses. This "puddle world task" is illustrated in Figure 3.3. Equipped with TD Learning, using a function approximator to learn the value function, could such an agent learn to avoid the puddles and move rapidly to the goal location, after substantial experience in the environment?

Boyan and Moore (1995) provided simulation evidence suggesting that such learning is impossible. They tried a variety of different value function approximators, including a backpropagation network with a single hidden layer, but none of them converged on a good solution to the problem. Indeed, as the agent continued to explore the environment, the estimate of future reward for locations kept changing, failing to settle down to a fixed value function.

This observation suggests that the difficulty in learning this problem arises from

a specific feature of reinforcement learning. In particular, the value function that the function approximator is trying to learn is, in a way, a moving target. Early in training, when the agent is unlikely to make it to the goal location, the expected future reward for a given location might be quite low. Later, if the agent has had some successes, the value for the same location might be higher. If the value function is stored in a large look-up table, then adjusting the value of one location has no influence on the values associated with other locations, allowing for small incremental changes in the value function. When using a function approximator, however, adjusting parameters (e.g., artificial neural network connection weights) for one location will likely change the value assigned to many other locations, potentially causing the un-learning of appropriate values for those other locations (catastrophic inference). This is a reasonable hypothesis for the observed lack of convergence.

In the following year, Sutton (1996) showed that this task could be learned by a TD Learning agent by hard-wiring the hidden layer units of the backpropagation network used to learn the value function to implement a fixed sparse conjunctive (coarse) code of the agent's location. The specific encoding used was one that had been previously proposed in the CMAC model of the cerebellum (Albus, 1975). Each hidden unit would become active only when the agent was in a location within a particular range of $x$ values *and* within a particular range of $y$ values. The conjoining of conditions on both coordinates is what made this code "conjunctive" in nature. Also, for any given location, only a small fraction of the hidden units displayed non-zero activity. This is what it means for the hidden representation to be a "sparse" code. Locations that were close to each other in the environment produced more overlap in the hidden units that were active than locations that were separated by a large distance. By ensuring that most hidden units had zero activity when connection weights were changed, this approach kept changes to the value function in one location from having a broad impact on the expected future reward at distant locations. (In the backpropagation of error learning algorithm, a connection weight is changed in proportion to the activity on the sending side of that connection, so there is no change if there is no activity being sent.) By engineering the hidden layer representation, this reinforcement learning problem was solved.

This is not a general solution, however. If the same approach was taken for another reinforcement learning problem, it is quite possible that the CMAC representation would not be appropriate. Thus, the method proposed by Sutton (1996) does not help us understand how TD Learning might flexibly learn a variety of reinforcement learning tasks. This approach requires prior knowledge of the kinds of internal representations of sensory state that are easily associated with expected future reward, and there are simple learning problems for which such prior knowledge is unavailable.

We hypothesize that the key feature of the Sutton (1996) approach is that it produces a sparse conjunctive code of the sensory state. Representations of this kind need not be fixed, however, but might be learned at the hidden layers of neural networks. There is substantial evidence that sparse representations are generated in the cortex by neurons that release the transmitter GABA (O'Reilly and Munakata,

2001) via lateral inhibition. Biologically inspired models of the brain show that, the sparse representation in the hippocampus can minimize the overlap of representations assigned to different cortical patterns. This leads to *pattern separation*, avoiding the catastrophic interference, but also supports *generalization* by modifying the synaptic connections so that these representations can later participate jointly in *pattern completion* (O'Reilly and McClelland, 1994; Noelle, 2008).

Computational cognitive neuroscience models have shown that a combination of feedforward and feedback inhibition naturally produces sparse conjunctive codes over a collection of excitatory neurons (O'Reilly and Munakata, 2001). Such patterns of lateral inhibition are ubiquitous in the mammalian cortex (Kandel et al., 2012). Importantly, neural networks containing such lateral inhibition can still learn to represent input information in different ways for different tasks, retaining flexibility while producing the kind of sparse conjunctive codes that may support reinforcement learning. Sparse distributed representation schemes have the useful properties of coarse codes while reducing the likelihood of interference between different representations.

## 3.3 Methods for Learning Sparse Representations

### 3.3.1 Lateral inhibition

Lateral inhibition can lead to sparse distributed representations (O'Reilly and Munakata, 2001) by making a small and relatively constant fraction of the artificial neurons active at any one time (e.g., 10% to 25%). Such representations achieve a balance between the generalization benefits of overlapping representations and the interference avoidance offered by sparse representations. Another way of viewing the sparse distributed representations produced by lateral inhibition is in terms of a balance between competition and cooperation between neurons participating in the representation.

It is important to note that sparsity can be produced in distributed representations by adding regularization terms to the learning loss function, providing a penalty during optimization for weights that cause too many units to be active at once (French, 1991; Zhang et al., 2015; Liu et al., 2018). This learning process is not necessary, however, when lateral inhibition is used to produce sparse distributed representations. With this method, feedforward and feedback inhibition enforce sparsity from the very beginning of the learning process, offering the benefits of sparse distributed representations even early in the reinforcement learning process.

### 3.3.2 k-Winners-Take-All mechanism

Computational cognitive neuroscience models have shown that fast pooled lateral inhibition produces patterns of activation that can be roughly described as $k$-Winners-Take-All ($k$WTA) dynamics (O'Reilly and Munakata, 2001). A $k$WTA function ensures that approximately $k$ units out of the $n$ total units in a hidden layer are strongly

active at any given time. Applying a $k$WTA function to the net input in a hidden layer gives rise to a sparse distributed representations, and this happens without the need to solve any form of constrained optimization problem. The $k$WTA function is provided in Algorithm 5. The $k$WTA mechanism only requires sorting the net input vector in the hidden layer in every feedforward direction to find the top $k+1$ active neurons. Consequently, it has at most $\mathrm{O}(n + k \log k)$ computational time complexity using a partial quicksort algorithm, where $n$ is the number of neurons in the largest hidden layer, and $k$ is the number of winner neurons. $k$ is relatively smaller than $n$. For example $k = 0.1 \times n$ is considered for the simulations reported in this chapter, and this ratio is commonly used in the literature.

**Algorithm 5** The $k$-Winners-Take-All Function

Input $\eta$: net input to the hidden layer
Input $k$: number of winner units
Input constant parameter $0 < q < 1$, e.g., $q = 0.25$
Find top $k+1$ most active neurons by sorting $\eta$, and store them in $\eta'$ in descending order
Compute $k$WTA bias, $b \leftarrow \eta'_k - q(\eta'_k - \eta'_{k+1})$
**return** $\eta_{kWTA} \leftarrow \eta - b$

### 3.3.3 Feedforward kWTA neural network

In order to bias a neural network toward learning sparse conjunctive codes for sensory state inputs, we constructed a variant of a backpropagation neural network architecture with a single hidden layer (see Figure 3.1) that utilizes the $k$WTA mechanism described in Algorithm 5.

Consider a continuous control task (such as grid-world or puddle-world) where the state of the agent is described as 2D coordinates, $s = (x, y)$. Suppose that the agent has to choose between four available actions $\mathcal{A}$ ={North, South, East, West}. Suppose that the $x$ coordinate and the $y$ coordinate are in the range $[0, 1]$ and each $x$ and $y$ range is discretized uniformly to $n_x$ and $n_y$ points correspondingly. Let's denote $\mathbf{X} = [0 : 1/n_x : 1]$, and $\mathbf{Y} = [0 : 1/n_y : 1]$ as the discretized vectors, i.e., $\mathbf{X}$ is a vector with $n_x+1$ elements from 0 to 1, and all points between them with the grid size $1/n_x$. To encode a coordinate value for input to the network, a Gaussian distribution with a peak value of 1, a standard deviation of $\sigma_x = 1/n_x$ for the $x$ coordinate, and $\sigma_y = 1/n_y$ for the $y$ coordinate, and a mean equal to the given continuous coordinate value, $\mu_x = x$, and $\mu_y = y$, was used to calculate the activity of each of the $\mathbf{X}$ and $\mathbf{Y}$ input units. Let's denote the results as $\mathbf{x}$ and $\mathbf{y}$. The input to the network is a concatenation vector $\mathbf{s} := (\mathbf{x}, \mathbf{y})$.

We calculate the *net input* values of hidden units based on the network inputs, i.e., the weighted sum of the inputs

$$\eta := W^{ih}\mathbf{s} + b^{ih}, \tag{3.1}$$

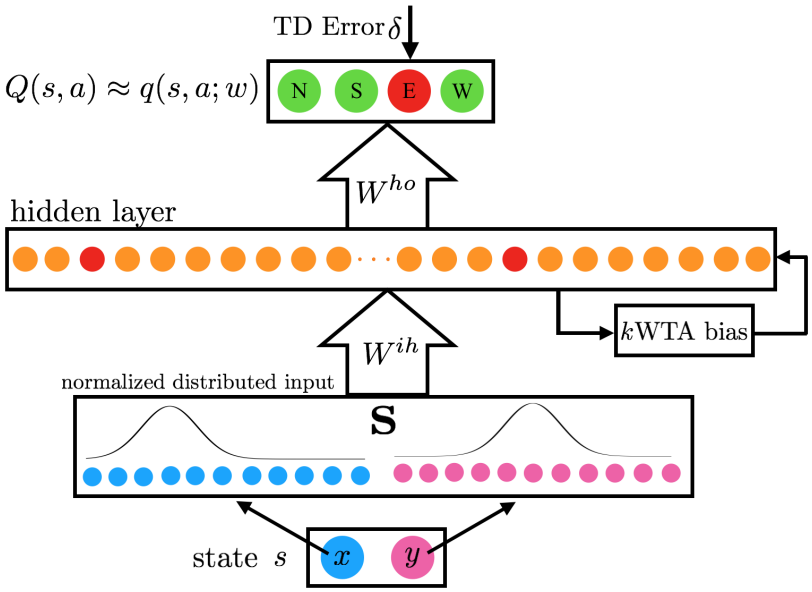



Figure 3.1: The $k$WTA neural network architecture: a backpropagation network with a single layer equipped with the $k$-Winner-Take-All mechanism (from Algorithm 5). The $k$WTA bias is subtracted from the hidden units net input that causes polarized activity which supports the sparse conjunctive representation. Only 10% of the neurons in the hidden layer have high activation. Compare the population of red (winner) neurons to the orange (loser) ones.

where $W^{ih}$ are weights, and $b^{ih}$ are biases from the input layer to the hidden layer.

After calculating the net input, we compute the $k$WTA bias, $b$, using Algorithm 5. We subtract $b$ from all of the net input values, $\eta$, so that the $k$ hidden units with the highest net input values have positive net input values, while all of the other hidden units' adjusted net input values become negative. These adjusted net input values, i.e. $\eta_{kwta} = \eta - b$ were transformed into unit activation values using a logistic sigmoid activation function (gain of 1, offset of $-1$), resulting in hidden unit activation values in the range between 0.0 and 1.0,

$$\mathbf{h} = \frac{1}{1 + e^{-(\eta_{kwta} - 1)}}, \tag{3.2}$$

with the top $k$ units having activations above 0.27 (due to the $-1$ offset), and the "losing" hidden units having activations below that value. The $k$ parameter controlled the degree of sparseness of the hidden layer activation patterns, with low values producing more sparsity (i.e., fewer hidden units with high activations). In the simulations in this chapter, we set $k$ to be 10% of the total number of hidden units. The output layer of the $k$WTA neural network is fully connected to the hidden layer and has $|\mathcal{A}|$ units, with each unit $i = 1, \ldots, |\mathcal{A}|$ representing the state-action values $q(s, a_i; w)$. We calculate these values by computing the activation in the output layer

$$\mathbf{q} = W^{ho}\mathbf{h} + b^{ho}, \tag{3.3}$$

where $W^{ho}$ are weights, and $b^{ho}$ are biases, from the hidden layer to the output layer. The output units used a linear activation function, hence, $\mathbf{q}$ is a vector of the state-action values.

In addition to encouraging sparse distributed representations, this $k$WTA mechanism has two properties that are worthy of note. First, introducing this highly nonlinear mechanism violates some of the assumptions relating the backpropagation of error procedure to stochastic gradient descent in error. Thus, the connection weight changes recommended by the backpropagation procedure may slightly deviate from those which would lead to local error minimization in this network. We opted to ignore this discrepancy, however, trusting that a sufficiently small learning rate would keep these deviations small. Second, it is worth noting that this particular $k$WTA mechanism allows for a distributed pattern of activity over the hidden units, making use of intermediate levels of activation. This provides the learning algorithm with some flexibility, allowing for a graded range of activation levels when doing so reduces network error. As connection weights from the inputs to the hidden units grow in magnitude, however, this mechanism will drive the activation of the top $k$ hidden units closer to 1 and the others closer to 0. Indeed, an examination of the hidden layer activation patterns in the $k$WTA-equipped networks used in this study revealed that the $k$ winning units consistently had activity levels close to the maximum possible value, once the learning process was complete.

## 3.4 Numerical Simulations

### 3.4.1 Experiment design

In order to assess our hypothesis that biasing a neural network toward learning sparse conjunctive codes for sensory state inputs will improve TD Learning when using a neural network as a function approximator for $q(s, a; w)$ state-action value function, we constructed three types of backpropagation networks:

**kWTA network.** A single layer backpropagation neural network equipped with the $k$-Winners-Take-All mechanism. See Figures 3.1 and 3.2(c).

**Regular network.** A single layer backpropagation neural network without the $k$WTA mechanism. See Figure 3.2(b).

**Linear network.** A linear neural network without a hidden layer. See Figure 3.2(a)

There was complete connectivity between the input units and the hidden units and between the hidden units and the output units. For the linear network, there was full connectivity between the input layer and the output layer. All connection weights were initialized to uniformly sampled random values in the range $[-0.05, 0.05]$.

In order to investigate the utility of sparse distributed representations, simulations were conducted involving three relatively simple reinforcement learning control

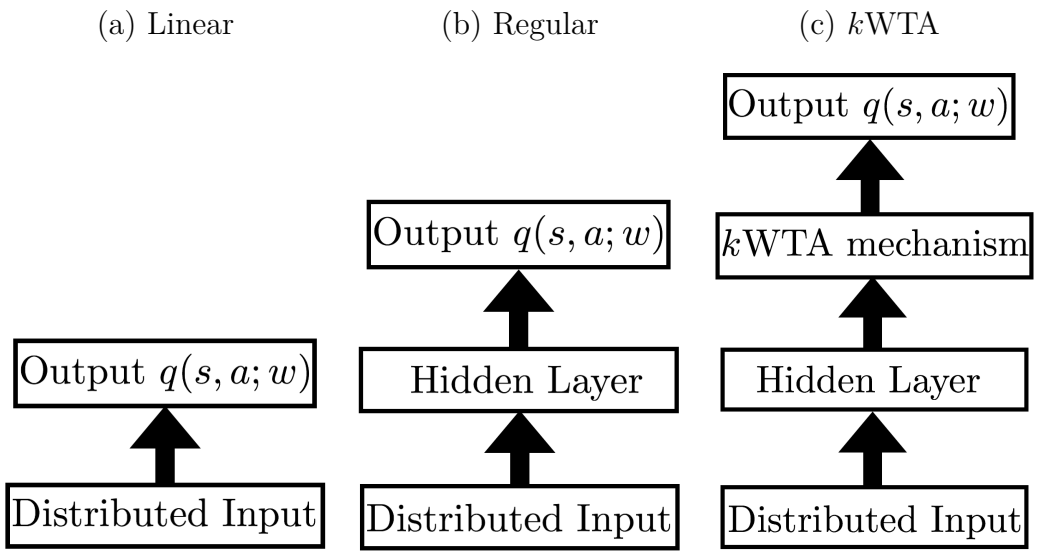



Figure 3.2: The neural network architectures used as the function approximator for state action values $q(s, a; w)$. (a) Linear network. (b) Regular backpropagation neural network. (c) $k$WTA network.

tasks: the *Puddle-world* task, the *Mountain-car* task, and the *Acrobot* task. These reinforcement learning problems were selected because of their extensive use in the literature (Boyan and Moore, 1995; Sutton, 1996). In this section, each task is described. We tested the SARSA variant of TD Learning (Sutton and Barto, 2017). (See Algorithm 6.) on each of the three neural networks architectures. The Matlab code for the simulations is available at `http://rafati.net/td-sparse/`.

The specific parameters for each simulation can be found in the description of the simulation tasks below. In section 3.5, the numerical results for training performance for each task are reported and discussed.

### 3.4.2 The Puddle-world task

The agent in the Puddle-world task attempts to navigate in a dark 2D grid world to reach a goal location at the top right corner, and it should avoid entering poisonous "puddle" regions (Figure 3.3). In every episode of Algorithm 1, the agent is located in a random state. The agent can choose to move in one of four directions, $\mathcal{A} = \{\text{North, South, East, West}\}$. For the Puddle-world task, the $x$-coordinate and the $y$-coordinate of the current state, $s$, were presented to the neural network over two separate pools of input units. Note that these coordinate values were in the range $[0, 1]$, as shown in Figure 3.3. Each pool of input units consisted of 21 units, with each unit corresponding to a coordinate location between 0 and 1, inclusive, in increments of 0.05. To encode a coordinate value for input to the network, a Gaussian distribution with a peak value of 1, a standard deviation of 0.05, and a mean equal to the given

**Algorithm 6** TD SARSA learning method using a neural network

Initialize parameters of $q(s, a; w)$ arbitrarily
**repeat** (for each episode)
  initialize $s$
  $a \leftarrow$`EPSILON-GREEDY`$(q(s, \mathcal{A}; w), \epsilon)$
  **repeat** (for each step of episode $t = 1, \ldots, T$)
    take action $a$, observe reward $r$ and next state $s'$
    $a' \leftarrow$`EPSILON-GREEDY`$(q(s', \mathcal{A}; w), \epsilon)$
    compute TD error, $\delta = r + \gamma q(s', a'; w) - q(s, a; w)$
    compute $\nabla L(w) = -\delta \nabla_w Q(s, a; w)$
    update weights (e.g. SGD step) $w \leftarrow w - \alpha \nabla L(w)$
    $s \leftarrow s'$, $a \leftarrow a'$
  **until** ($s$ is terminal)
**until** (convergence or reaching to max number of episodes)

continuous coordinate value was used to calculate the activity of each of the 21 input units (see Figure 3.1).

We use each of the three mentioned neural network models as function approximators for state-action values (see Figure 3.2). All networks had four output units, with each output corresponding to one of the four directions of motion. The hidden layer for both regular BP and $k$WTA neural networks had 220 hidden units. In the $k$WTA network, only 10% (or 22) of the hidden units were allowed to be highly active.

At the beginning of the simulation, the exploration probability, $\epsilon$, was set to a relatively high value of 0.1, and it remained at this value for much of the learning process. Once the average magnitude of $\delta$ over an episode fell below 0.2, the value of $\epsilon$ was reduced by 0.1% each time the goal location was reached. Thus, as the agent became increasingly successful at reaching the goal location, the exploration probability, $\epsilon$, approached zero. (Annealing the exploration probability is commonly done in systems using TD Learning.) The agent continued to explore the environment, one episode after another, until the average absolute value of $\delta$ was below 0.01 and the goal location was consistently reached, or a maximum of $44,100$ episodes had been completed. This value was heuristically selected as a function of the size of the environment: $(21 \times 21) \times 100 = 44,100$. Each episode of SARSA was terminated if the agent had reached the goal in the corner of the grid or after the maximum steps, $T = 80$, had been taken. The learning rate remained fixed $\alpha = 0.005$ during the training.

When this reinforcement learning process was complete, we examined both the behavior of the agent and the degree to which its value function approximations, $q(s, a; w)$, matched the correct values determined by running SARSA to convergence while using a large look-up table to capture the value function.

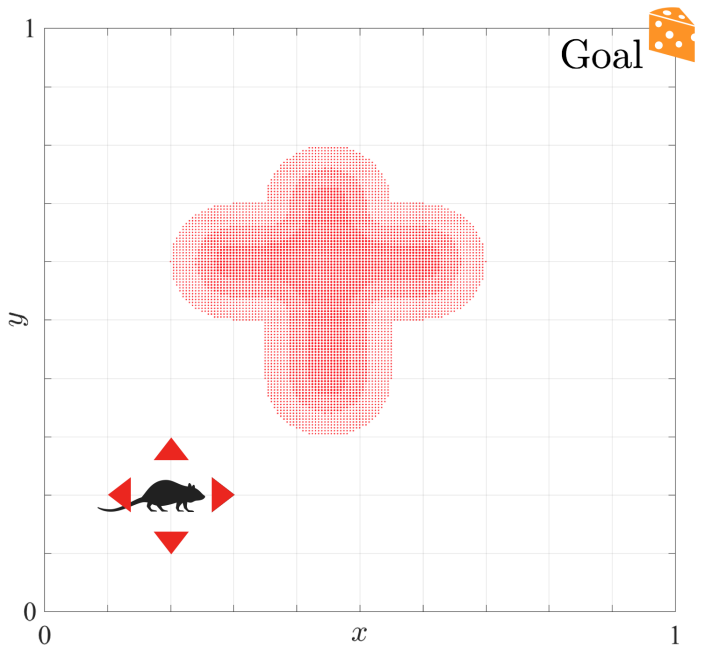



Figure 3.3: The agent in puddle-world task attempts to reach the goal location (fixed in the Northeast corner) in the least time steps by avoiding the puddle. The agent moves a distance of 0.05 either North, South, East, or West on each time step. Entering a puddle produces a reward of $(-400 \times d)$, where $d$ is the distance of the current location to the edge of the puddle. This value was $-1$ for most of the environment, but it had a higher value, 0, at the goal location in the Northeast corner. Finally, the agent receives a reward signal of $-2$ if it had just attempted to leave the square environment. This pattern of reinforcement was selected to parallel that previously used in Sutton (1996).

### 3.4.3 The Mountain-car task

In this reinforcement learning problem, the task involves driving a car up a steep mountain road to a high goal location. The task is difficult because the force of gravity is stronger than the car's engine (see Figure 3.4). In order to solve the problem, the agent must learn first to move away from the goal, then use the stored potential energy in combination with the engine to overcome gravity and reach the goal state. The state of the Mountain-car agent is described by the car's position and velocity, $s = (x, \dot{x})$. There are three possible actions: $\mathcal{A} = \{$`forward, neutral, backward`$\}$ throttle of the motor. After choosing action $a \in \mathcal{A}$, the car's state is updated by the following equations

$$x_{t+1} = \text{bound}(x_t + \dot{x}_{t+1}) \tag{3.4a}$$

$$\dot{x}_{t+1} = \text{bound}(\dot{x} + 0.0001a_t - 0.0025\cos(3x_t)), \tag{3.4b}$$

where the bound function keeps state variables within their limits: $x \in [-1.2, +0.5]$ and $\dot{x} \in [-0.07, +0.07]$. If the car reaches the left environment boundary, its velocity

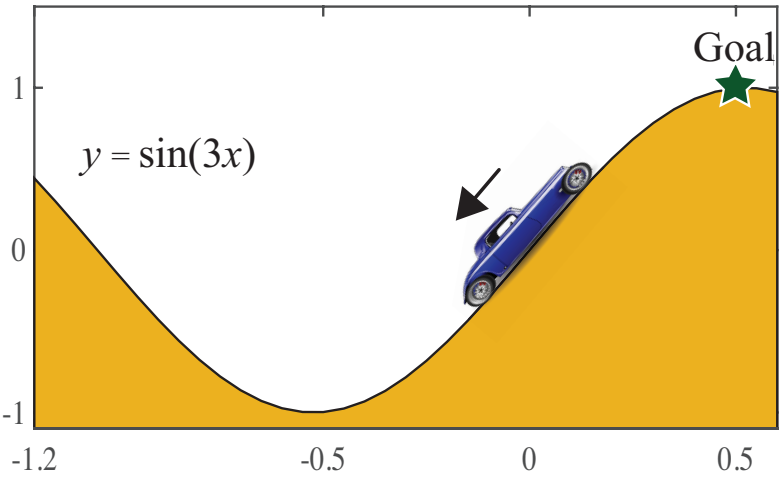



Figure 3.4: The Mountain-car task. The goal is to drive an underpowered car up a steep hill. The agent received -1 reward for each time step until it reached the goal, at which point it received 0 reward.

is reset to zero. In order to present the values of the state variables to the neural network, each variable was encoded as a Gaussian activation bump surrounding the input corresponding to the variable value. Every episode started from a random position and velocity. Both the $x$-coordinate and the $\dot{x}$ velocity was discretized to 60 mesh points, allowing each variable to be represented over 61 inputs. To encode a state value for input to the network, a Gaussian distribution was used to calculate the activity of each of the 61 input units for that variable. The network had a total of 122 inputs.

All networks had three output units, each corresponding to one of the possible actions. Between the 122 input units and the 3 output units was a layer of $61 \times 61 \times 0.7 = 2604$ hidden units (for the $k$WTA and the Regular networks). The hidden layer of the $k$WTA network was subject to the previously described the $k$WTA mechanism, parameterized so as to allow 10%, or 260, of the hidden units to be highly active.

We used the SARSA version of TD Learning (Algorithm 1). The exploration probability, $\epsilon$, was initialized to 0.1 and it was decreased by 1% after each episode until reaching a value of 0.0001. The agent received a reward of $r = -1$ for most of the states, but it received a reward of $r = 0$ at the goal location ($x \geq 0.5$). If the car collided with the leftmost boundary, the velocity was reset to zero, but there was no extra punishment for bumping into the wall. The learning , $\alpha = 0.001$ stayed fixed during the learning. The agent explored the Mountain-car environment in *episodes*. Each episode began with the agent being placed at a location within the environment, sampled uniformly at random. Actions were then taken, and connection weights updated, as described above. The episode ended when the agent reached the goal location or after the maximum of $T = 3000$ actions had been taken. The agent continued to explore the environment, one episode after another, until the average

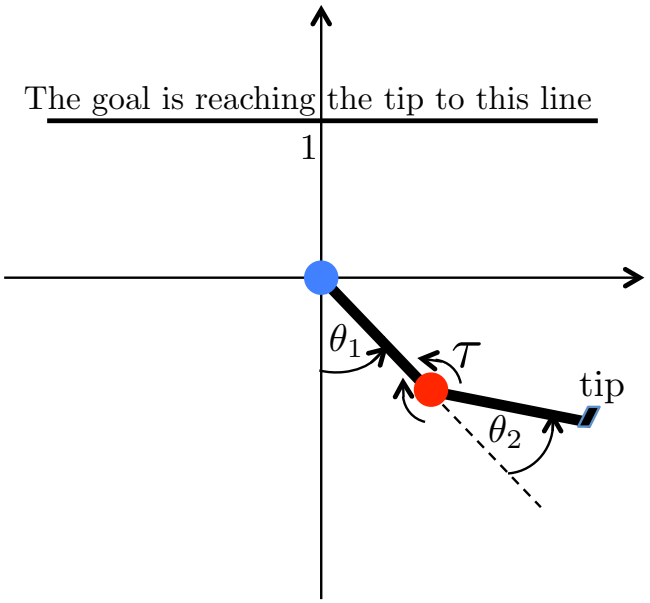



Figure 3.5: The Acrobot task. The goal is to swing the tip (i.e. "feet") above the horizontal by the length of the lower "leg" link. The agent receives -1 reward until it reaches to goal, at which point it receives 0 reward.

absolute value of $\delta$ was below 0.05 and the goal location was consistently reached, or a maximum of $200,000$ episodes had been completed.

### 3.4.4 The Acrobot task

We also examined the utility of learning sparse distributed representations on the Acrobot control task, which is a more complicated task and one attempted by Sutton (1996). The *acrobot* is a two-link under-actuated robot, a simple model of a gymnast swinging on a high-bar (Sutton, 1996). (See Figure 3.5.) The state of the acrobot is determined by four continuous state variables: two joint angles $(\theta_1, \theta_2)$ and corresponding velocities. Thus, the state the agent can be formally described as $s = (\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2)$. The goal is to control the acrobot so as to swing the end tip ("feet") above the horizontal by the length of the lower "leg" link. Torque may only be applied to the second joint. The agent receives $-1$ reward until it reaches the goal, at which point it receives a reward of 0. The frequency of action selection is set to 5 Hz, and the time step, $\Delta t = 0.05s$, is used for numerical integration of the equations describing the dynamics of the system. A discount factor of $\gamma = 0.99$ is used. The

equations of motion for Acrobot are,

$$\ddot{\theta}_1 = -d_1^{-1}(d_2\ddot{\theta}_2 + \phi_2), \tag{3.5a}$$

$$\ddot{\theta}_2 = -\left(m_2 l_{c2}^2 + I_2 - \frac{d_2^2}{d_1}\right)^{-1}\left(\tau + \frac{d_2}{d_1}\phi_1 - \phi_2\right), \tag{3.5b}$$

where $d_1$, $d_2$, $\phi_1$ and $\phi_2$ are defined as

$$d_1 = m_1 l_{c1}^2 + m_2 l_1 l_{c2}\cos(\theta_2) + I_2, \tag{3.6a}$$

$$d_2 = m_2\left(l_{c2}^2 + l_1 l_{c2}\cos(\theta_2)\right) + I_2, \tag{3.6b}$$

$$\begin{aligned}\phi_1 = &-m_2 l_1 l_{c2}\dot{\theta_1}^2\sin(\theta_2) - 2m_2 l_1 l_{c2}\dot{\theta}_2\dot{\theta}_1\sin(\theta_2) \\ &+ (m_1 l_{c1} + m_2 l_1)g\cos(\theta_1 - \pi/2) + \phi_2,\end{aligned} \tag{3.6c}$$

$$\phi_2 = m_2 l_{c2} g\cos(\theta_1 + \theta_2 - \pi/2). \tag{3.6d}$$

The agent chooses between three different torque values, $\tau \in \{-1, 0, 1\}$, with torque only applied to the second joint. The angular velocities are bounded to $\theta_1 \in [-4\pi, 4\pi]$ and $\theta_2 \in [-9\pi, 9\pi]$. The values $m_1 = 1$ and $m_2 = 1$ are the masses of the links, and $l_1 = 1m$ and $l_2 = 1m$ are the lengths of the links. The values $l_{c1} = l_{c1} = 0.5m$ specify the location of the center of mass of links, and $I_1 = I_2 = 1kg\ m^2$ are the moments of inertia for the links. Finally, $g = 9.8m/s^2$ is the acceleration due to gravity. These physical parameters are previously used in Sutton (1996).

In our simulation, each of the four state dimensions is divided over 20 uniformly spaced ranges. To encode a state value for input to the network, a Gaussian distribution with a peak value of 1, a standard deviation of 1/20, and a mean equal to the given continuous state variable value was used to calculate the activity of each of the 21 inputs for that variable. Thus, the network had 84 total inputs.

The network had three output units, each corresponding to one of the possible values of torque applied: clockwise, neutral, or counter-clockwise. Between the 84 inputs and the 3 output units was a layer of 8400 hidden units for the $k$WTA network and the regular backpropagation network. For the $k$WTA network, only 10%, or 840, of the hidden units were allowed to be highly active.

The Acrobot agent explores its environment in *episodes*. Each episode of learning starts with the Acrobot agent hanging straight down and at rest (i.e., $s = (0, 0, 0, 0)$). The episode ends when the agent reaches the goal location or after the maximum of $T = 2000$ actions are taken. At the beginning of a simulation, the exploration probability, $\epsilon$, is set to a relatively high value of 0.05. When the agent first reaches the goal, the value of $\epsilon$ starts to decrease, being reduced by 0.1% each time the goal location is reached. Thus, as the agent becomes increasingly successful at reaching the goal location, the exploration probability, $\epsilon$, approaches to lower bound of 0.0001. The agent continues to explore the environment, one episode after another, until the average absolute value of $\delta$ is below 0.05 and the goal location is consistently reached, or a maximum of $200,000$ episodes are completed. A small learning rate $\alpha = 0.0001$ was used and stayed fixed during the learning.

## 3.5 Results and Discussions

We compared the performance of our $k$WTA neural network with that produced by using a standard backpropagation network (Regular) with identical parameters. We also examined the performance of a linear network (Linear), which had no hidden units but only complete connections from all input units directly to the four output units (see Figure 3.2).

### 3.5.1 The puddle-world task

Figure 3.6(a)-(c) show the learned value function (plotted as $\max_a Q(s, a)$ for each location, $s$, for representative networks of each kind. Also, Figure 3.6(d)-(f) display the learned policy at the end of learning. Finally, we show learning curves displaying the episode average value of the TD Error, $\delta$, over episodes in Figure 3.6(g)-(i).

In general, the Linear network did not consistently learn to solve this problem, sometimes failing to reach the goal or choosing paths through puddles. The Regular backpropagation network performed much better, but its value function approximation still contained sufficient error to produce a few poor action choices. The $k$WTA network, in comparison, consistently converged on good approximations of the value function, almost always resulting in optimal paths to the goal.

For each network, we quantitatively assessed the quality of the paths produced by agents using the learned value function approximations. For each simulation, we froze the connection weights (or, equivalently, set the learning rate to zero), and we sequentially produced an episode for each possible starting location, except for locations inside of the puddles. The reward accumulated over each episode was recorded, and, for each episode that ended at the goal location, we calculated the sum squared deviation of these accumulated reward values from that produced by an optimal agent (identified using SARSA with a large look-up table for the value function). The mean of this squared deviation measure, over all successful episodes, was recorded for each of the 20 simulations run for each of the 3 network types. The means of these values, over 20 simulations, are displayed in Figure 3.7. The backpropagation network had significantly less error than the linear network ($t(38) = 4.692$; $p < 0.001$), and the $k$WTA network had significantly less error than the standard backpropagation network ($t(38) = 6.663$; $p < 0.001$). On average, the $k$WTA network deviated from optimal performance by less than one reward point.

We also recorded the fraction of episodes which succeeded at reaching the goal for each simulation run. The mean rate of goal attainment, across all non-puddle starting locations, for the linear network, the backpropagation network, and the $k$WTA network were 93.3%, 99.0%, and 99.9%, respectively. Despite these consistently high success rates, the linear network exhibited significantly more failures than the backpropagation network ($t(38) = 2.306$; $p < 0.05$), and the backpropagation network exhibited significantly more failures than the $k$WTA network ($t(38) = 2.205$; $p < 0.05$).

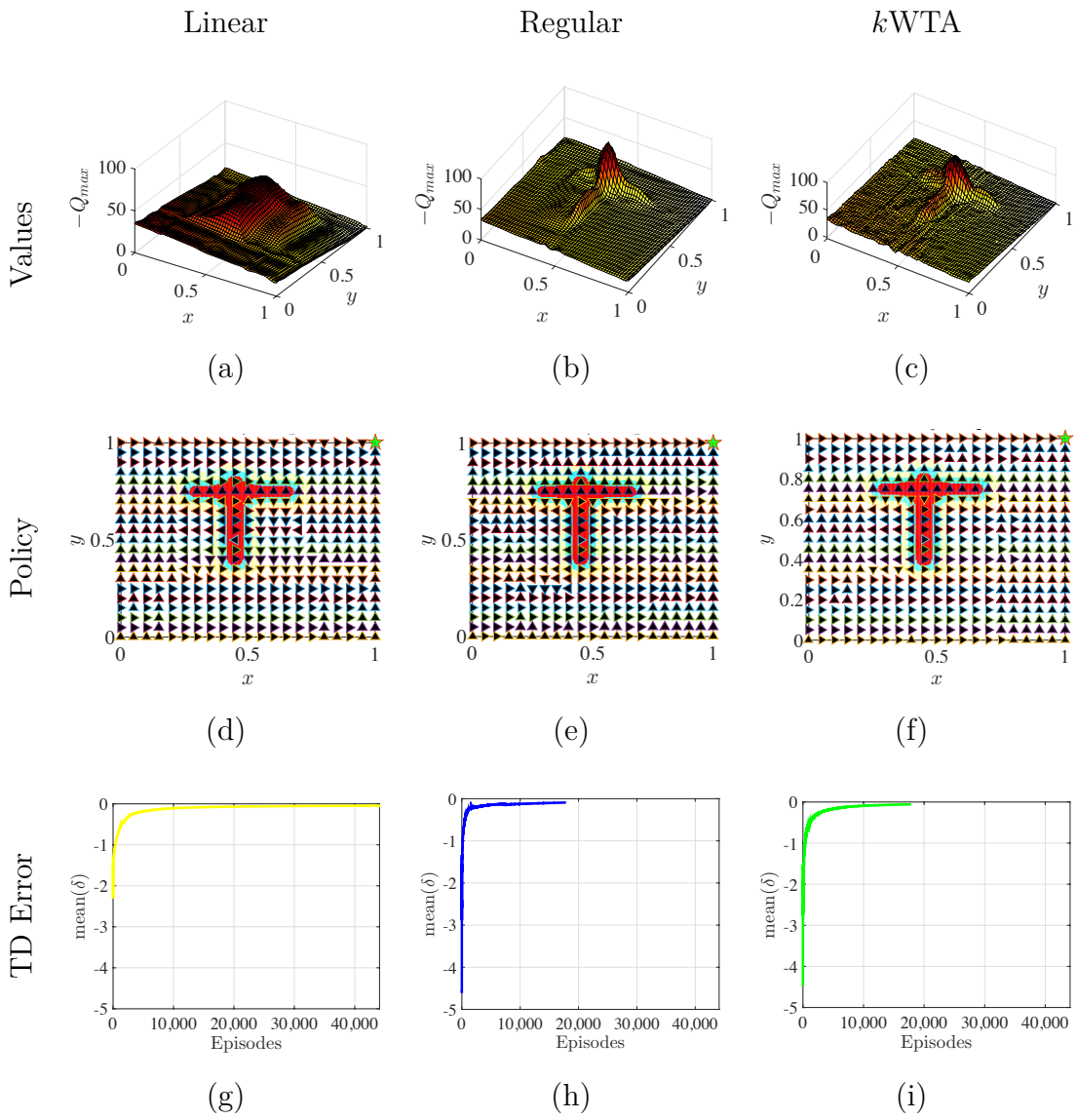



Figure 3.6: The performance of various learned value function approximators may be compared in terms of their success at learning the true value function, the resulting action selection policy, and the amount of experience in the environment needed to learn. The approximate of the state values, expressed as $max_a\ Q(s, a)$ for each state, $s$ is given. (a) Values of states for the Linear network. (b) Values of states for the Regular network. (c) Values of state for $k$WTA network. The actions selected at a grid of locations is shown in the middle row. (d) Policy of states derived from Linear network. (e) Policy of states derived from Regular network. (f) Policy of states derived from $k$WTA network. The learning curve, showing the TD Error over learning episodes, is shown on the bottom. (g) Average TD error for training Linear network. (h) Average TD error for training Regular network. (i) Average TD error for training $k$WTA network.

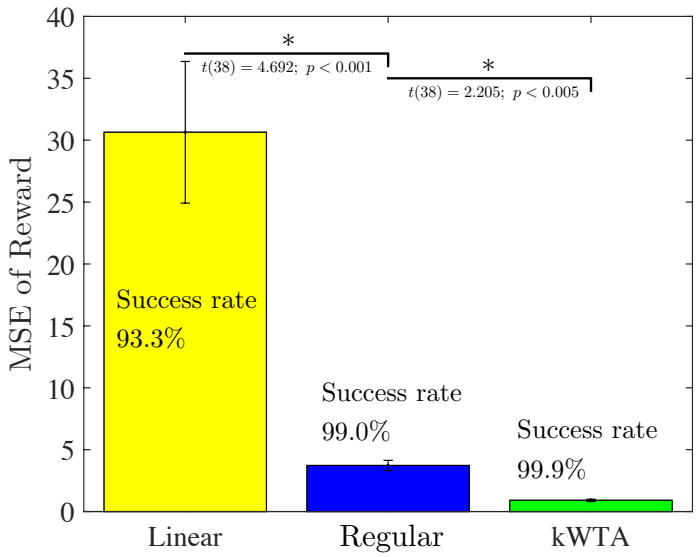



Figure 3.7: Averaged over 20 simulations of each network type, these columns display the mean squared deviation of accumulated reward from that of optimal performance. Error bars show one standard error of the mean.

### 3.5.2 The mountain-car task

Figure 3.8 (a)-(c) show the value function (plotted as $\max_a Q(s, a)$ for each location, $s$) for representative networks of each kind. In the middle row of Figure 3.8(d)-(f), we show the learning curves, displaying the episode average value of the TD Error, $\delta$, over training episodes, and also the number of time steps needed to reach the goal during training episodes. In the last row, we display the results of testing the performance of the networks. The test performances were collected after every epoch of 1000 training episodes, and there were no changes to the weights and no exploration during the testing phase. The value function for the $k$WTA network is the closest numerically to optimal $Q$-table results, and the policy after training for the $k$WTA network is the most stable.

### 3.5.3 The Acrobot task

In the first row of Figure 3.9(a)-(c), the episode average value of the TD Error, $\delta$, over training episodes, and also the number of time steps needed to reach the goal during training episodes are shown for different networks during the training phase. In the last row of Figure 3.9(d)-(f), we display the results of testing performance of the networks for the three network architectures. At testing time, any changes to the weights were avoided, and there was no exploration.

From these results, we can see that only the $k$WTA network could learn the optimal policy. Both the Linear network and the Regular backpropagation network failed to learn the optimal policy for the Acrobot control task.

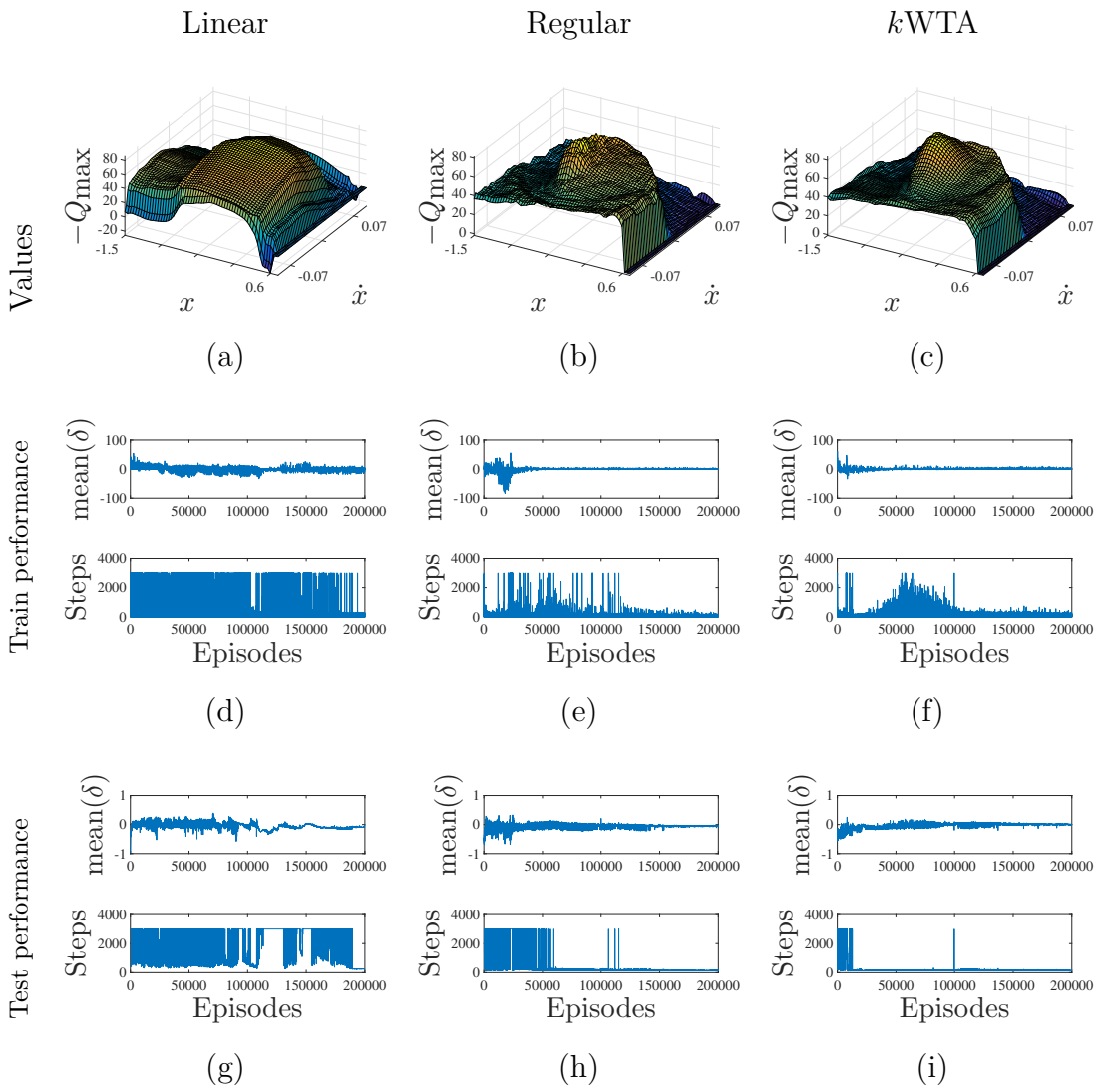



Figure 3.8: The performance of various networks trained to perform the Mountain-car task. The top row contains the approximate value of states after training ($\max_a Q(s,a)$). (a) Values approximated from Linear network. (b) Values approximated from Regular BP network. (c) Values approximated from $k$WTA network. The middle row shows two statistics over training episodes: the top subplot shows the average value of the TD Error, $\delta$, and the bottom subplot shows the number of time steps during training episodes. Average of TD Error and total steps in training for (d) Linear (e) Regular (f) $k$WTA is given. The last row reports the testing performance, which was measured after each epoch of 1000 training episodes. During the test episodes the weight parameters were frozen and no exploration was allowed. Average of TD Error and the total steps for (g) Linear (h) Regular (i) $k$WTA are given.

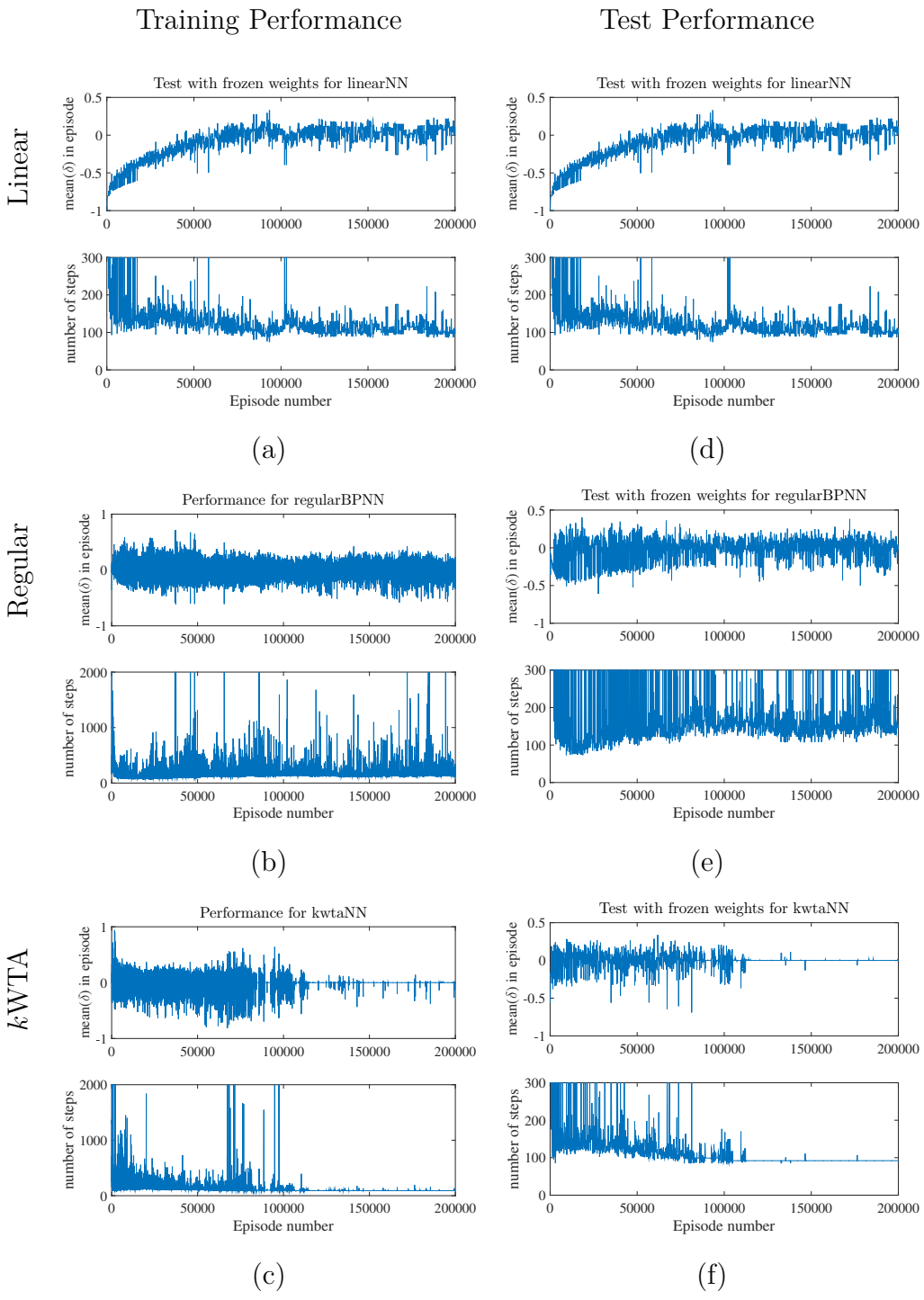



Figure 3.9: The performance of various networks trained to solve the Acrobot control task. The top row results correspond to training performance of the SARSA Algorithm 1. The average TD Error (top subplot) and the total number of steps (bottom subplot) for each episode of learning are given for (a) Linear (b) Regular and (c) $k$WTA neural networks are given. The bottom row are the performance results for the testing that was measured after every epoch of 1000 training episodes in which the weight parameters were frozen and no exploration was allowed. Average of TD error and total steps for (d) Linear (e) Regular (f) $k$WTA are given.

## 3.6 Future Work

Using sparse conjunctive representation of the agent's state not only can help in the solving of the simple reinforcement learning tasks, but it might also help improve the learning of some large-scale tasks, too. In the future, we will extend this work to the deep reinforcement learning framework, where the value function is approximated by a deep Convolutional Neural Network (CNN). The $k$WTA mechanism can be used in the fully connected layers of the CNN in order to generate sparse representations using the lateral inhibition like mechanism. We anticipate that employing the $k$WTA mechanism to generate sparse conjunctive representations will lead to faster and more robust learning in complicated deep RL tasks.

A deep CNN equipped with a $k$-Winners-Take-All mechanism in the fully connected layers can also be used for supervised learning. This is particularly interesting in applications such as image recognition, when two images look similar (they share similar features), but they belong to different classes. We hypothesize that the conjunctive sparse representation generated by $k$WTA can learn subtle differences, where regular CNN may fail to learn.

## 3.7 Conclusions

Inspired by the lateral inhibition that appears in cortical areas, we implemented a state-action value function approximator that utilizes a $k$-Winners-Take-All mechanism (O'Reilly, 2001). The proposed method solves the previously impossible-to-solve control tasks. The simulation results demonstrate that a mechanism for learning sparse conjunctive codes for the agent's sensory state can help overcome learning problems observed when using TD Learning with a value function approximator (Rafati and Noelle, 2015, 2017). Artificial neural networks can be forced to produce such sparse codes over their hidden units by including a process akin to the sort of pooled lateral inhibition that is ubiquitous in the cerebral cortex. In this way, these simulation results both support our method for improving reinforcement learning and also lend preliminary support to the hypothesis that the midbrain dopamine system, along with associated circuits in the basal ganglia, do, indeed, implement a form of TD Learning, and the observed problems with TD learning do not arise in the brain due to the encoding of sensory state information in circuits that make use of lateral inhibition.

# Chapter 4

# Learning Representations in Model-Free Hierarchical Reinforcement Learning


## Abstract

Common approaches to Reinforcement Learning (RL) are seriously challenged by large-scale applications involving huge state spaces and sparse delayed reward feedback. Hierarchical Reinforcement Learning (HRL) methods attempt to address this scalability issue by learning action selection policies at multiple levels of *temporal abstraction*. Abstraction can be had by identifying a relatively small set of states that are likely to be useful as subgoals, in concert with the learning of corresponding skill policies to achieve those subgoals. Many approaches to *subgoal discovery* in HRL depend on the analysis of a model of the environment, but the need to learn such a model introduces its own problems of scale. Once subgoals are identified, skills may be learned through *intrinsic motivation*, introducing an internal reward signal marking subgoal attainment. In this chapter, we present a novel model-free method for subgoal discovery using incremental unsupervised learning over a small memory of the most recent experiences (trajectories) of the agent. When combined with an intrinsic motivation learning mechanism, this method learns both subgoals and skills, based on experiences in the environment. Thus, we offer an original approach to HRL that does not require the acquisition of a model of the environment, suitable for large-scale applications. We demonstrate the efficiency of our method on two RL problems with sparse delayed feedback: a variant of the rooms environment and the first screen of the ATARI 2600 *Montezuma's Revenge* game.

## 4.1 Introduction

The reinforcement learning problem suffers from serious scaling issues. Methods such as transfer learning (Ammar et al., 2012; Singh, 1992; Taylor and Stone, 2009), and Hierarchical Reinforcement Learning (HRL) attempt to address these issues (Barto and Mahadevan, 2003; Hengst, 2010; Dayan and Hinton, 1992; Dietterich, 2000). HRL is an important computational approach intended to tackle problems of scale by learning to operate over different levels of *temporal abstraction* (Sutton et al., 1999; Parr and Russell, 1997; Krishnamurthy et al., 2016; Stolle and Precup, 2002). The acquisition of hierarchies of reusable skills is one of the distinguishing characteristics of biological intelligence (Botvinick et al., 2009; Diuk et al., 2013; Badre et al., 2010), and the learning of such hierarchies is an important open problem in computational reinforcement learning. The development of robust HRL methods will provide a means to acquire relevant knowledge at multiple levels of abstraction, potentially speeding learning and supporting generalization.

A number of approaches to HRL have been suggested. One approach focuses on action sequences, subpolicies, or "options" that appear repeatedly during the learning of a set of tasks (Sutton et al., 1999; Levy and Shimkin, 2011; Fox et al., 2017; Bacon et al., 2017; Stolle and Precup, 2002; Bakker and Schmidhuber, 2004). Such frequently reused subpolicies can be abstracted into skills that can be treated as individual actions at a higher level of abstraction. A somewhat different approach to temporal abstraction involves identifying a set of states that make for useful *subgoals* (Goel and Huber, 2003; Simsek et al., 2005; McGovern and Barto, 2001; Machado and Bowling, 2016). This introduces a major open problem in HRL: that of *subgoal discovery*.

A variety of researchers have proposed approaches to identifying useful subpolicies and reifying them as skills (Pickett and Barto, 2002; Thrun and Schwartz, 1995; Mannor et al., 2004; Stolle and Precup, 2002). For example, Sutton et al. (1999) proposed the *options framework*, which involves abstractions over the space of actions. At each step, the agent chooses either a one-step "primitive" action or a "multi-step" action policy (an option). Each option defines a policy over actions (either primitive or other options) and comes to completion according to a termination condition. Other researchers have focused on identifying subgoals — states that are generally useful to attain — and learning a collection of skills that allow the agent to efficiently reach those subgoals. Some approaches to subgoal discovery maintain the value function in a large look-up table (Sutton et al., 1999; Goel and Huber, 2003; Simsek et al., 2005; McGovern and Barto, 2001), and most of these methods require building the state transition graph, providing a model of the environment and the agents possible interactions with it (Machado et al., 2017; Simsek et al., 2005; Goel and Huber, 2003; Mannor et al., 2004; Stolle and Precup, 2002). Formally, the state transition graph is a directed graph $G = (V, E)$ with a set of vertices, $V \subseteq \mathcal{S}$ and set of edges $E \subseteq \mathcal{A}(\mathcal{S})$, where $\mathcal{S}$ is the set of states and $\mathcal{A}(\mathcal{S})$ is the set of allowable actions when in a given state. Since actions typically modify the state of the agent, each

directed edge, $(s, s') \in E$, indicates an action that takes the agent from state $s$ to state $s'$. In nondeterministic environments, a probability distribution over subsequent states, given the current state and an action, $p(s'|s, a)$, is maintained as part of the model of the environment. One method of this kind that was applied to a somewhat larger scale task — the first screen of the ATARI 2600 game called *Montezuma's Revenge* — is that of Machado and Bowling (2016). This method constructs the combinatorial transition graph Laplacian matrix, and an eigen-decomposition of that matrix produces candidate subgoals. While it was shown that some of these candidates make for useful subgoals, only heuristic domain-sensitive methods have been reported for identifying useful subgoals from the large set of candidates (e.g., thousands). Thus, previously proposed subgoal discovery methods have provided useful insights and have been demonstrated to improve learning, but there continue to be challenges with regard to scalability and generalization. Scaling to large state spaces will generally mandate the use of some form of nonlinear function approximator to encode the value function, rather than a look-up table. More importantly, as the scale of a reinforcement learning problem increases, the tractability of obtaining a good model of the environment, capturing all relevant state transition probabilities, precipitously decreases.

Once useful subgoals are discovered, an HRL agent should be able to learn the skills to attain those subgoals through the use of *intrinsic motivation* — artificially rewarding the agent for attaining selected subgoals (Singh et al., 2010; Vigorito and Barto, 2010). In such systems, knowledge of the current subgoal is needed to estimate future intrinsic reward, resulting in value functions that consider subgoals along with states (Vezhnevets et al., 2017). The nature and origin of "good" intrinsic reward functions is an open question in reinforcement learning, however, and a number of approaches have been proposed. Singh et al. (2010) explored agents with intrinsic reward structures in order to learn generic options that can apply to a wide variety of tasks. Value functions have also been generalized to consider goals along with states (Vezhnevets et al., 2017). Such a parameterized universal value function, $q(s, g, a; w)$, integrates the value functions for multiple skills into a single function taking the current subgoal, $g$, as an argument.

Recently, Kulkarni et al. (2016) proposed a scheme for temporal abstraction that involves simultaneously learning options and a hierarchical control policy in a deep reinforcement learning framework. Their approach does not use separate $Q$-functions for each option, but, instead, treats the option as an argument. This method lacks a technique for automatic subgoal discovery, however, forcing the system designer to specify a set of promising subgoal candidates in advance. The approach proposed in this chapter is inspired by Kulkarni et al. (2016), which has advantages in terms of scalability and generalization, but it incorporates automatic subgoal discovery.

It is important to note that *model-free* HRL, which does not require a model of the environment, still often requires the learning of useful internal representations of states. When learning the value function using a nonlinear function approximator, such as a deep neural network, relevant features of states must be extracted in order

to support generalization at scale. A number of methods have been explored for learning such internal representations during model-free RL (Tesauro, 1995; Rafati and Noelle, 2017; Mnih et al., 2015), and deep model-based HRL (Kulkarni et al., 2016; Li et al., 2017; Lyu et al., 2019). However, selecting the right representation is still an open problem (Maillard et al., 2011).

In this chapter, we seek to address major open problems in the integration of internal representation learning, temporal abstraction, automatic subgoal discovery, and intrinsic motivation learning, all within the model-free HRL framework (Rafati and Noelle, 2019a). We propose and implement efficient and general methods for subgoal discovery using unsupervised learning and anomaly (outlier) detection (Rafati and Noelle, 2019c). These methods do not require information beyond that which is typically collected by the agent during model-free reinforcement learning, such as a small memory of recent experiences (agent trajectories). Our methods are fundamentally constrained in three ways, by design. First, we remain faithful to a model-free reinforcement learning framework, eschewing any approach that requires the learning or use of an environment model. Second, we are devoted to integrating subgoal discovery with intrinsic motivation learning, and temporal abstraction. Specifically, we conjecture that intrinsic motivation learning can increase appropriate state space coverage, supporting more efficient subgoal discovery. Lastly, we focus on subgoal discovery algorithms that are likely to scale to large reinforcement learning tasks. The result is a unified model-free HRL algorithm that incorporates the learning of useful internal representations of states, automatic subgoal discovery, intrinsic motivation learning of skills, and the learning of subgoal selection by a "meta-controller". We demonstrate the effectiveness of this algorithm by applying it to a variant of the rooms task (illustrated in Figure 4.12(a)), as well as the initial screen of the ATARI 2600 game called *Montezuma's Revenge* (illustrated in Figure 4.17(a)).

## 4.2 Failure of RL in Tasks with Sparse Feedback

In an RL problem, the agent should implement a policy, $\pi$, from states, $\mathcal{S}$, to possible actions, $\mathcal{A}$, to maximize its expected return from the environment (Sutton and Barto, 2017). At each cycle of interaction, the agent receives a state, $s$, from the environment, takes an action, $a$, and one time step later, the environment sends a reward, $r \in \mathbb{R}$, and an updated state, $s'$. Each cycle of interaction, $e = (s, a, r, s')$ is called a transition *experience*. The goal is to find an optimal policy that maximizes the expected value of the return, i.e. the cumulative sum of future rewards, $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'+1}$, where $\gamma \in [0, 1]$ is a discount factor, and $T$ as a final step. It is often useful to define a parametrized value function $Q(s, a; w)$ to estimate the expected value of the return. As described in Chapter 2, Q-learning is a Temporal Difference (model-free RL) algorithm that attempts to find the optimal value function by minimizing the loss

function, $L(w)$, which is defined over a recent *experience memory*, $\mathcal{D}$:

$$L(w) \triangleq \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\Big(r + \gamma \max_{a'} Q(s', a'; w) - Q(s, a; w)\Big)^2\Big]. \tag{4.1}$$

Learning representations of the value function is challenging for tasks with sparse, and delayed rewards, since in (4.1), $r = 0$ (or an undiagnostic constant value such as $r = -1$) for most experiences. Even if the agent accidentally visits a rare rewarding state, where $r > 0$, the experience replay mechanism often fails to learn the value of those states (Mnih et al., 2015).

Another major problem in RL is the exploration-exploitation trade-off. Common approaches, such as the $\epsilon$-greedy method, are not sufficiently efficient in exploring the state space to succeed on large-scale complex problems (Bellemare et al., 2016; Vigorito and Barto, 2010). As a simple example, consider the task of navigation in the *4-room environment with a key and a car*, shown in Figure 4.7 The agent is rewarded for entering the grid square containing the key, and it is more substantially rewarded for entering the grid square with the car after obtaining the key. The other states are not rewarded. Learning even this simple task is challenging for a reinforcement learning agent.

Our intuition, shared with other researchers, is that hierarchies of abstraction will be critical for successfully solving problems of this kind. To be successful, the agent should represent knowledge at multiple levels of spatial and temporal abstraction. Appropriate abstraction can be had by identifying a relatively small set of states that are likely to be useful as *subgoals* and jointly learning the corresponding skills of achieving those subgoals, using intrinsic motivation.

## 4.3 Hierarchical Reinforcement Learning

In Hierarchical Reinforcement Learning, a central goal is to allow learning to happen simultaneously on several levels of abstraction. As a simple illustration of the problem, consider the task of navigation in the *4-room environment with a key and a car*.

The 4-room is a grid-world environment, consisting of 4 rooms, as shown in Figure 4.1. These rooms are connected through *doorways*. The agent receives the most reward if it navigates in this environment, finds a key, picks up the key, and moves to a car. The agent is initialized in an arbitrary location in an arbitrary room. The location of the key, the car, and the doorways are arbitrary and can vary. This is a variant of the *rooms* task introduced by (Sutton et al., 1999). The agent receives $r = +10$ reward for reaching the key and $r = +100$ if it moves to the car while carrying the key. The agent can move either $\mathcal{A}$ ={North, South, East, West} on each time step. Bumping to the wall boundaries is punished with a reward of $r = -2$. There is no reward or punishment for exploring the space. Learning in this environment with sparse feedback is challenging for a reinforcement learning agent. To successfully generalize to different environment configurations, the agent should represent knowledge at multiple levels of spatiotemporal abstraction. It should also

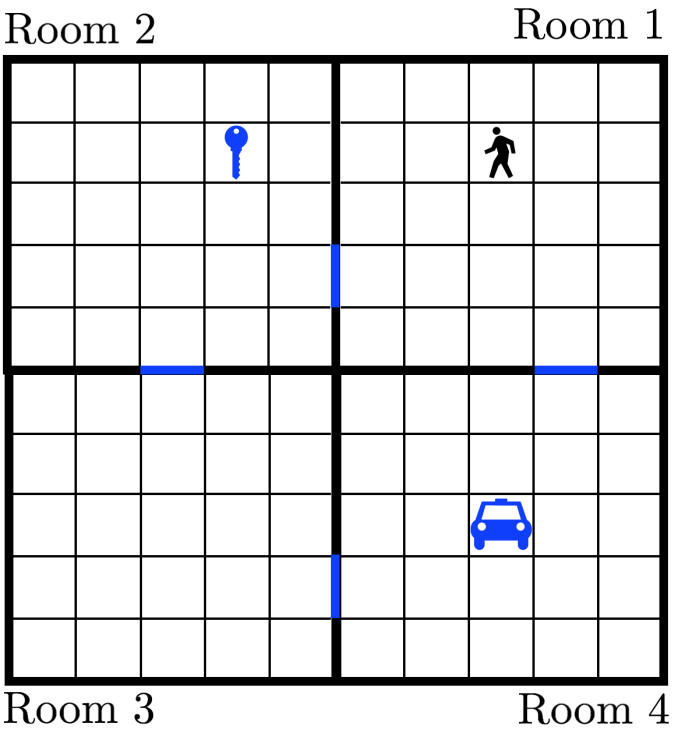


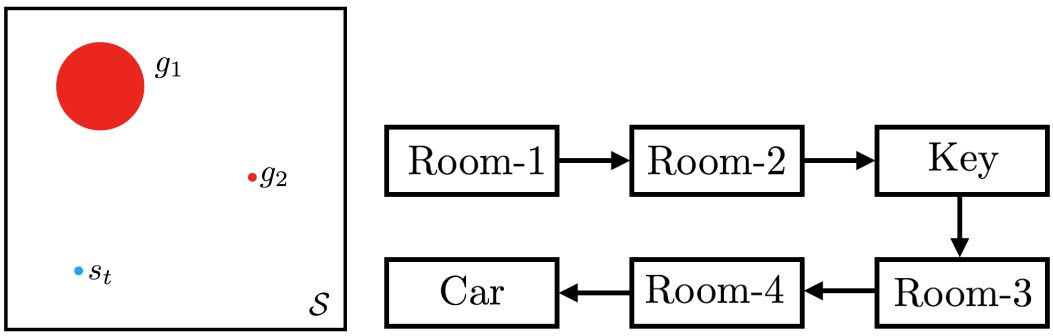

Figure 4.1: The rooms task with a key and a car. The agent should explore the *rooms* to first find the key and then find the car. The key and the car can be in any of the 4 rooms in any arbitrary locations. The agent moves either $\mathcal{A} = \{North, South, East, West\}$ on each time step. The agent receives $r = +10$ reward for getting the key and $r = +100$ if it reaches the car with the key. The blue objects on the map — doorways, key, and car — are useful *subgoals*.

learn to explore the environment efficiently. The quality of the agent's *policy* depends critically on the location of the doorways, the key, and the car.

### 4.3.1 Subgoals vs. Options

Learning to obtain a subgoal is typically easier than learning the full task. Pursuing a *subtask* of "go to room 2", which is part of the solution to the full task, is much easier than the 4-rooms task, itself. A *subgoal*, $g$, is a state that must be visited as part of pursuing a major *goal*. The subspace $\mathcal{G} \subseteq \mathcal{S}$ is called the *goal space*, and the members of $\mathcal{G}$ are the candidate subgoals (or goals) that the RL agent might pursue to solve the task. In this task, a good set of subgoals is $\mathcal{G} =$ `{doorways, key, car}`. However, being able to pursue other sets of states can be useful in learning the task. For example, learning how to move from room 2 to room 1 can be a useful skill, and successful execution of this skill is marked by moving to a subset region of state space. (See Figure 4.2.)

In some of the HRL literature the term "option" is used to describe a temporally




(a) (b)

Figure 4.2: (a) The state space $\mathcal{S}$ and the state of the agent $s_t$. The intrinsic goal can be either reaching from $s_t$ to a region or set of states, $g_1 \subset \mathcal{S}$, or to a single state $g_2 \in \mathcal{S}$. (b) An option is a transition from a set of states to another set of states.

abstracted action (Machado and Bowling, 2016; Fox et al., 2017). The literature can be confusing, however, as other researchers use this term to describe a subgoal — a specific state in the state space. In this chapter, we use the former notion of option. An option, $o_{ij}$, is a transition from one set of states, $g_i$, to another, $g_j$. For example, going from room 1 to room 2 can be considered an option (i.e., an extended action).

### 4.3.2 Spatiotemporal Hierarchies

The rooms task has at least two types of hierarchical structure.

1. *Spatial Hierarchy.* The states have similarity structure in terms of belonging to a certain room. This could be captured as a hierarchical relationship between locations and rooms. For example, at the moment captured in Figure 4.1, the key is located in relative location $(x_{key}, y_{key})$ in room 2, and the agent is in relative location $(x_{agent}, y_{agent})$ in room 1.

2. *Temporal Hierarchy.* To solve the task, the agent first needs to get the key, and, after accomplishing this subgoal, the agent should move to the car. Thus, the temporal order of subgoals is another dimension of hierarchy.

### 4.3.3 Hierarchical Reinforcement Learning Subproblems

The rooms task has both clear skills and clear subgoals. To accomplish each temporal subgoal $g \in$ `{doorways, key, car}`, the agent needs to go to the corresponding location of $g$. (See Figure 4.3.) Learning how to explore the state space to reach any arbitrary location in the given room is a valuable skill. This skill could be reused to reach any of a number of subgoals. This is often called *spacing* and it can be accomplished through *intrinsic motivation*.

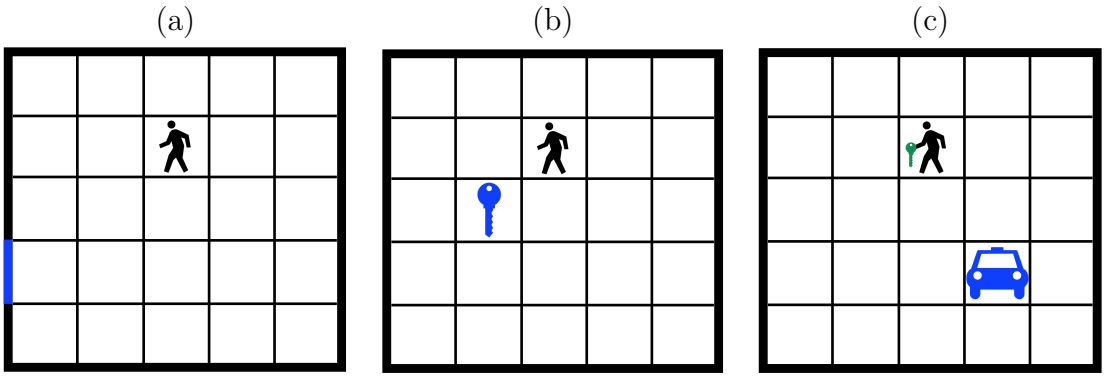



Figure 4.3: The rooms task requires the agent to be able to navigate and reach a certain subgoal, $g$, from its current state. (a) Moving to a *doorway*. (b) Moving to the key. (c) Moving to the car. Learning how to space the state space through intrinsic motivation can facilitate learning.

One approach to solving the hierarchical reinforcement learning problem is to break down the problem into the following three subproblems:

**Subproblem 1: Learning a meta-policy to choose a subgoal.** Learning to operate at different levels of abstraction is essential in hierarchical reinforcement learning. In this subproblem, the purpose is training a top-level controller to learn an optimal meta-policy to choose a proper subgoal, $g_t$, form set of candidate subgoals, $\mathcal{G}$, and deliver it to the lower-level controller. We refer to the top-level learner as the *meta-controller*, and the lower-level learner is just called the *controller* (Kulkarni et al., 2016). Formally, the objective is to find a mapping $\Pi : \mathcal{S} \rightarrow \mathcal{G}$, which ideally is an optimal meta-policy that maximizes the return. For example, in the rooms task (Figure 4.1), it is expected that the meta-controller chooses a proper subgoal (i.e. room 2) for the agent located in room 1.

**Subproblem 2: Exploring the state space while learning subtaks through intrinsic motivation.** Intrinsic motivation refers to learning behavior that is driven by internal rewards. A reinforcement learning agent can be intrinsically motivated to explore its environment and learn about the effects of its actions. The skills learned during this period of exploration are then reused to great effect later to solve many unfamiliar problems very quickly. The agent is assigned to solve a task of reaching to a subgoal, $g_t$. Formally, the agent should learn an optimal policy $\pi(s_t, g_t)$ for all possible (available) states, $s_t \in \mathcal{S}$ and for all subgoals, $g_t \in \mathcal{G}$. In particular, we present algorithms for intrinsically motivated hierarchical exploration for temporal difference learning. An example of intrinsic motivation is given in Figure 4.3. The meta-controller (top-level learner) assigns a goal, $g_t$, to the controller (lower-level learner), and the controller should learn how to reach to different locations in the room. Consequently, the agent learns how to navigate in the state space to reach an

arbitrary goal.

**Subproblem 3: Subgoal discovery.** In order to solve subproblems 1 and 2, a proper set of subgoals, $\mathcal{G}$, should be available. This require solving the subgoal discovery problem which is one of major open problems in the hierarchical reinforcement learning literature. Formally, we are interested in studying methods of learning to discover the proper candidate subgoals, $\mathcal{G}$, from the agent's past experiences memory $\mathcal{D}$. For example, a proper set of subgoals for the rooms task (see Figure 4.1) include the location of doorways, the key and the car. When learning begins, the subgoals are arbitrary, but once they are assigned to the controller, more experiences can be gathered through the process of intrinsic motivation learning. We introduce an unsupervised learning method that can discover the underlying structure in the experience space and use the learned representation to discover a good set of subgoals. Automatic subgoal discovery in model-free hierarchical reinforcement learning is an open problem that is addressed in the proposed HRL framework.

## 4.4 Meta-controller/Controller Framework

A straightforward computational approach for temporal abstraction is proposed by Kulkarni et al. (2016) in the meta-controller/controller framework. The agent in this framework makes decisions at two levels of abstraction:

(a) The top level module (*meta-controller*) takes the state, $s_t$, as input and picks a new subgoal, $g_t$.

(b) The lower level module (*controller*) uses both the state (or a meta-state $\tilde{s}_t$) and the chosen subgoal to select actions, continuing to do so until either the subgoal is reached or the episode is terminated.

If a subgoal is reached, the meta-controller then chooses another subgoal, and the above steps (a-b) repeat. In this chapter, we focus on only two levels of hierarchy, but the proposed methods can be expanded to greater hierarchical depth without loss of generality.

As shown in Figure 4.4, the agent uses a two-level hierarchy consisting of a controller and a meta-controller. At time step $t$, the meta-controller receives a state observation, $s = s_t$, from the environment. It has a policy for selecting a *subgoal*, $g = g_t$, from a set of subgoals, $\mathcal{G}$. The controller then selects actions in an effort to attain the given subgoal. The objective function for the controller is to maximize cumulative future intrinsic reward

$$\tilde{G}_t = \sum_{t'=t}^{t+T} \gamma^{t'-t} \tilde{r}_t(g), \tag{4.2}$$

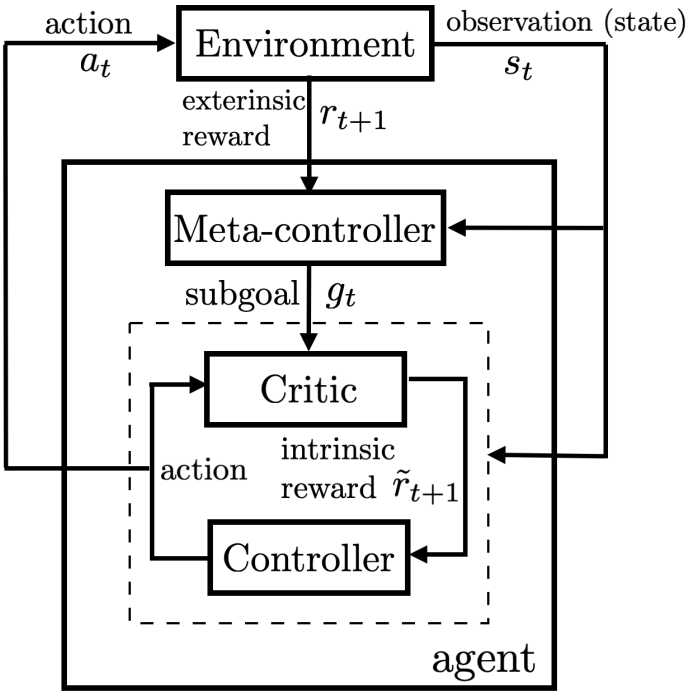



Figure 4.4: The Meta-Controller/Controller framework for temporal abstraction. The agent produces actions and receives sensory observations. Separate networks are used inside the meta-controller and controller. The meta-controller looks at the raw states and produces a policy over goals by estimating the value function $Q(s_t, g_t)$ (by maximizing expected future extrinsic reward). The controller takes states as input, along with the current goal, $(s_t, g_t)$, and produces a policy over actions by estimating the value function $q(s_t, g_t, a_t)$ to accomplish the goal $g_t$ (by maximizing expected future intrinsic reward). The internal critic checks if a goal is reached and provides an appropriate intrinsic reward to the controller. The controller terminates either when the episode ends or when $g_t$ is accomplished. The meta-controller then chooses a new subgoal, and the process repeats. This architecture is adapted from Kulkarni et al. (2016).

where $T$ is the maximum length of internal episodes to accomplish the subtask of reaching to subgoal $g$. Similarly, the objective of the meta-controller is to maximize the cumulative extrinsic reward

$$G_t = \sum_{t'=t}^{\mathcal{T}} \gamma^{t'-t} r_t, \tag{4.3}$$

where $\mathcal{T}$ is a final step. We can use two different $Q$ functions to learn policies for the controller and the meta-controller. The controller estimates the following $Q$ values

$$q(s, g, a) = \mathbb{E}_{\pi_{ag}} \Big[ \tilde{G}_t | s_t = s, g_t = g, a_t = a \Big], \tag{4.4}$$

where $g$ is the given subgoal in state $s$ and $\pi_{ag} = P(a|s,g)$ is the action policy. Similarly, the meta-controller estimates the following $Q$ values

$$Q(s,g) = \mathbb{E}_{\pi_g}\Big[G_t | s_t = s, g_t = g\Big], \tag{4.5}$$

where $\pi_g$ is the policy over subgoals. For example the optimal meta-policy for the rooms task is depicted in Figure 4.2 (b). It's important to note that the meta-controller experiences transitions, $(s_t, g_t, G_{t:t+T}, g_{t+T})$, at a slower time-scale than the controller's transitions, $(s_t, a_t, g_t, \tilde{r}_t, s_{t+1})$. Note that $G_{t:t+T}$ is the return (cumulative external reward) in Equation 4.3 for one episode of the controller with a length $T$.

In our implementation, the policy arises from estimating the value of each subgoal, $Q(s,g;\mathcal{W})$, and selecting the goal of highest estimated value. With the current subgoal selected, the controller uses its policy to select an action, $a \in \mathcal{A}$, based on the current state, $s$, and the current subgoal, $g$. In our implementation, this policy involves selecting the action that results in the highest estimate of the controller's value function, $q(s,g,a;w)$. Actions continue to be selected by the controller while an internal critic monitors the current state, comparing it to the current subgoal, and delivering an appropriate *intrinsic reward*, $\tilde{r}$, to the controller on each time step. Each transition experience, $(s,g,a,\tilde{r},s')$, is recorded in the controller's experience memory set, $\mathcal{D}_1$, to support learning. When the subgoal is attained, or a maximum amount of time has passed, the meta-controller observes the resulting state, $s_{t'} = s_{t+T+1}$, and selects another subgoal, $g' = g_{t+T+1}$, at which time the process repeats, but not before recording a transition experience for the meta-controller, $(s,g,G,s_{t'})$ in the meta-controller's experience memory set, $\mathcal{D}_2$. The parameters of the value function approximators are adjusted based on the collections of recent experiences.

For training the meta-controller value function, we minimize a loss function based on the reward received from the environment:

$$\mathcal{L}(\mathcal{W}) \triangleq \mathbb{E}_{(s,g,G,s_{t'})\sim\mathcal{D}_2}\Big[\Big(G + \gamma \max_{g'} Q(s_{t'}, g'; \mathcal{W}) - Q(s,g;\mathcal{W})\Big)^2\Big], \tag{4.6}$$

where $G = \sum_{t'=t}^{t+T} \gamma^{t'-t} r_{t'}$ is the accumulated external reward (return) between the selection of consecutive subgoals. The term, $\mathcal{Y} = G + \gamma \max_{g'} Q(s_{t'}, g'; \mathcal{W})$, in (4.6) is the target value for the expected return at the time that the meta-controller selected subgoal $g$. The controller improves its subpolicy, $\pi(a|s,g)$, by learning its value function, $q(s,g,a;w)$, over the set of recorded transition experiences. The controller updates its value function approximator parameters, $w$, so as to minimize its loss function:

$$L(w) \triangleq \mathbb{E}_{(s,g,a,\tilde{r},s')\sim\mathcal{D}_1}\Big[\Big(\tilde{r} + \gamma \max_{a'} q(s', g, a'; w) - q(s,g,a;w)\Big)^2\Big]. \tag{4.7}$$

The hierarchical reinforcement learning algorithm for meta-controller and controller learning is given in Algorithm 7.

**Algorithm 7** Meta-Controller and Controller Learning

Specify Subgoals space $\mathcal{G}$
Initialize $w$ in $q(s, g, a; w)$
Initialize $\mathcal{W}$ in $Q(s, g; \mathcal{W})$.
Initialize experience memories $\mathcal{D}_1$ and $\mathcal{D}_2$
**for** episode $= 1, \ldots, M$ **do**
  Initialize state $s_0 \in \mathcal{S}$, $s \leftarrow s_0$
  $G \leftarrow 0$
  $g \leftarrow$ `EPSILON-GREEDY`$(Q(s, \mathcal{G}; \mathcal{W}), \epsilon_2)$
  **repeat** for each step $t = 1, \ldots, T$
    compute $q(s, g, a; w)$
    $a \leftarrow$ `EPSILON-GREEDY`$(q(s, g, \mathcal{A}; w), \epsilon_1)$
    Take action $a$, observe $s'$ and external reward $r$
    Compute intrinsic reward $\tilde{r}$ from internal critic
    Store controller's intrinsic experience, $(s, g, a, \tilde{r}, s')$ to $\mathcal{D}_1$
    Sample $J_1 \subset \mathcal{D}_1$ and compute $\nabla L$
    Update controller's parameters, $w \leftarrow w - \alpha_1 \nabla L$
    Sample $J_2 \subset \mathcal{D}_2$ and compute $\nabla \mathcal{L}$
    Update meta-controller's parameters, $\mathcal{W} \leftarrow \mathcal{W} - \alpha_2 \nabla \mathcal{L}$
    $s \leftarrow s'$, $G \leftarrow G + r$
    Decay exploration rate of controller $\epsilon_1$
  **until** $s$ is terminal or subgoal $g$ is attained
  Decay exploration rate of meta-controller $\epsilon_2$
  Store meta-controller's experience, $(s_0, g, G, s')$ to $\mathcal{D}_2$
**end for**

## 4.5 Intrinsic Motivation Learning

Intrinsic motivation learning is the core idea behind the learning of value functions in the meta-controller and the controller. In some tasks with sparse delayed feedback, a standard RL agent cannot effectively explore the state space so as to have a sufficient number of rewarding experiences to learn how to maximize rewards. In contrast, the intrinsic critic in our HRL framework can send much more regular feedback to the controller, since it is based on attaining subgoals, rather than ultimate goals. As an example, our implementation typically awards an intrinsic reward of $+1$ when the agent attains the current subgoal, $g$, and $-1$ for any other state transition. Successfully solving a difficult task not only depends on such an intrinsic motivation learning mechanism, but also on the meta-controller's ability to learn how to choose the right subgoal for any given state, $s$, from a set of candidate subgoals. Indeed, identifying a good set of candidate subgoals is an additional prerequisite for success.

Developing skills through intrinsic motivation has at least two benefits: (1) exploration of large scale state spaces, and (2) enabling the reuse of skills in varied environments. Navigation in the rooms task requires the agent to learn the skills

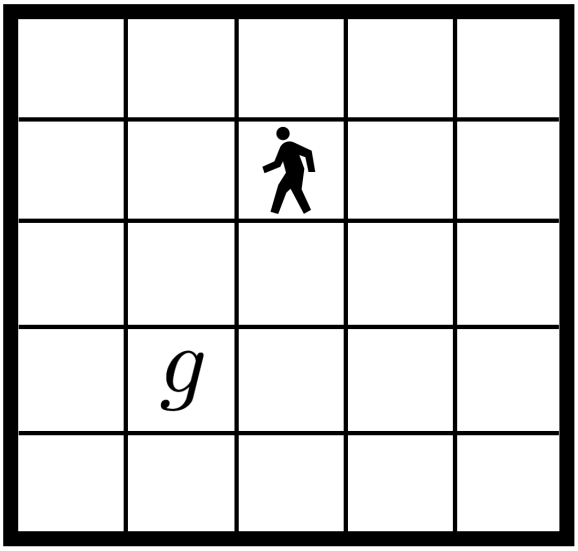



Figure 4.5: Grid-world task with a dynamic goal. At beginning of each episode an oracle chooses an arbitrary goal, $g \in \mathcal{S}$. The agent is initialized in a random location. On each time step, the agent has four action choices, $\mathcal{A} = \{$North, South, East, West$\}$. The agent receives $\tilde{r} = +1$ reward for successful episodes, reaching the goal, $g$. Bumping into the wall produces a reward of $\tilde{r} = -2$. There is no external reward or punishment from the environment for exploring the space.

to reach the doorways, key, and car (see Figure 4.3). These skills are acquired by learning to achieve subgoals that are provided by the meta-controller. The spacing of the state space can be done as a pretraining step or simultaneously with meta-policy training. In any case, a goal should be provided to the controller. This goal can be a random state or a region of state space (see Figure 4.2)). For now, we assume that the subgoal, $g \in \mathcal{G}$, is provided by an oracle (standing in for the meta-controller), and we focus only on learning to achieve this subgoal. The controller receives the state, $s_t$, and subgoal, $g_t$, as inputs and takes an action, $a_t$. This results in the sensing of the next state and the receipt of an intrinsic reward signal, $\tilde{r}_{t+1}$, from the internal critic. (See Figure 4.4.) The subgoal, $g_t$, remains the same for some time, $T$. There have been some studies concerning the appropriate structure of the intrinsic reward. We use the following form

$$\tilde{r}_{t+1} = \begin{cases} \min(r_{t+1}, -1) & \text{if } s_{t+1} \text{ is not terminal} \\ +1 & \text{if } s_{t+1} \text{ achieves the goal, } g_t \end{cases} \tag{4.8}$$

Other intrinsic reward functions might be considered. Indeed, the nature and origin of good intrinsic reward functions is an open question in reinforcement learning. As an attempt to solve Subproblem 1, we try to solve the task of navigation in a grid-world given a subgoal location, $g$. (See Figure 4.5.) For intrinsic motivation we can define a reward function like that above in Equation 4.8. The algorithm for intrinsic motivation learning for a random meta-controller is given in Algorithm 8.

**Algorithm 8** Intrinsic Motivation Learning

Specify Subgoals space $\mathcal{G}$
Initialize $w$ in $q(s, g, a; w)$
Initialize controller's experience memory, $\mathcal{D}_1$
Initialize agent's experience memory, $\mathcal{D}$
**for** episode $= 1, \ldots, M$ **do**
  Initialize state $s_0 \in \mathcal{S}$, $s \leftarrow s_0$
  Select a random subgoal $g$ from $\mathcal{G}$
  **repeat** for each step $t = 1, \ldots, T$
    compute $q(s, g, a; w)$
    $a \leftarrow$`EPSILON-GREEDY`$(q(s, g, \mathcal{A}; w), \epsilon_1)$
    Take action $a$, observe $s'$ and external reward $r$
    Compute intrinsic reward $\tilde{r}$ from internal critic
    Store controller's intrinsic experience, $(s, g, a, \tilde{r}, s')$ to $\mathcal{D}_1$
    Store agent's experience, $(s, a, s', r)$ to $\mathcal{D}$
    Sample $J_1 \subset \mathcal{D}_1$ and compute $\nabla L$
    Update controller's parameters, $w \leftarrow w - \alpha_1 \nabla L$
    $s \leftarrow s'$
    Decay exploration rate of controller $\epsilon_1$
  **until** $s$ is terminal or subgoal $g$ is attained
**end for**

## 4.6 Experiment on Intrinsic Motivation Learning

Here, we want to show why intrinsic motivation can be useful through transferring knowledge and reusing skills. It is important to note that, we trained an agent in a single room (a grid-world environment) to learn the navigation task (see Figure 4.5). In the original rooms task in Figure 4.1, there are walls between rooms and specific doorways between rooms. We assume that there is an oracle in the meta-controller that provides the subgoals and a transformation from the real state to the input state to the state-goal network in Figure 4.6. We make this assumption here for the purpose of simplicity in the implementation and in order to prove the advantages of the intrinsic motivation in hierarchical reinforcement learning. We will revisit the intrinsic motivation problem once again in Section 4.8 and solve the rooms task using a Unified Model-Free HRL framework, without having access to the mentioned oracle. For now, let's assume that there is an oracle (instead of a meta-controller) that can transform the real external state of the agent in the rooms task into the location of the agent in the current room, providing this transformed location as the state-goal network input. This makes it possible to reuse the learned navigation skill to solve the rooms task (which is consists of four rooms).

### 4.6.1 Training the State-Goal Value Function

Here, we introduce the neural network architecture that we use to approximate $q(s, g, a; w)$ while training the controller depicted in Figure 4.6.

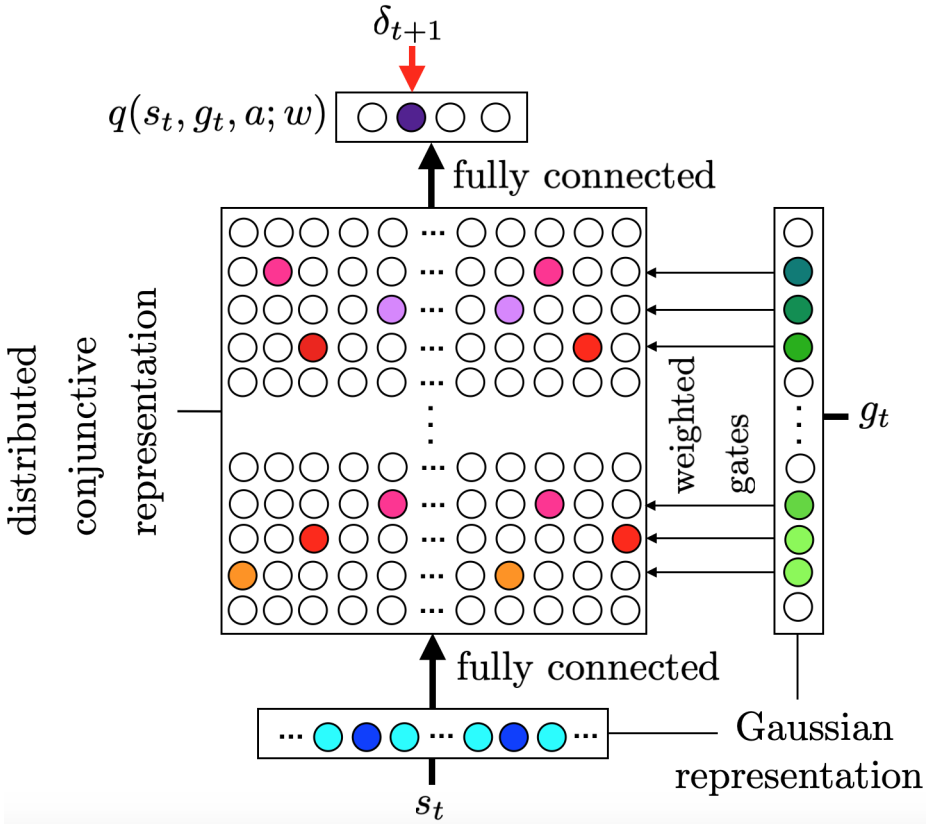



Figure 4.6: The state-goal neural network architecture used to approximate the value function for the controller, $q(s, g, a; w)$. The function takes the state, $s_t$, and the goal, $g_t$, as inputs. The first layer produces the Gaussian representation separately for $s_t$ and $g_t$. The state representation is connected fully to the hidden layer, and the $k$-Winners-Take-All mechanism produces a sparse representation for $s_t$. The goal representation is connected only to the corresponding row of units. We assume that an oracle in the meta-controller transform the state in rooms task to a proper state for the state-goal netwrok that is trained on the navigation in the gridworld (single room) task.

The controller $Q$-function, $q(s, g, a; w)$, takes the state, $s_t$, and the goal, $g_t$, as inputs. The first layer produces the Gaussian representation separately for $s_t$ and $g_t$. Let's denote the Gaussian representation of $s_t$ by $\hat{s}_t$ and the goal one by $\hat{g}_t$. The Gaussian representation for $s_t$ is similar to the one discussed in Chapter 3. The state input, $\hat{s}_t$, is connected fully to the hidden layer, with the connection weight matrix being $w^{(1)}$. The $k$-Winners-Take-All mechanism (10% of hidden units) produces a sparse representation for the state, $s_t$. The subgoal input, $\hat{g}_t$, is connected to the

hidden layer with a gate layer. This mechanism was included in hopes of avoiding catastrophic interference during reinforcement learning. The hidden layer is connected fully to the output units, with the weight matrix being $w^{(2)}$. The network is trained in a manner similar to standard backpropagation, with a forward pass determining activations and a backward pass performing error credit assignment. This training process is summarized in Algorithm 9. The grid-world room is discretized into $5 \times 5$ windows. The number of the hidden units in each row of $w^{(1)}$ is 50 and the total number of hidden units are 1250 (considering that there are 25 columns for each row). We assume that the oracle standing in for the meta-controller handles the transformation of the state in the rooms task to the input for the state-goal network.

**Algorithm 9** Forward pass, and backpropagation for network in Figure 4.6

**Forward pass.** Computing $q(s, g, a; w)$

initialize $q^{output} \leftarrow$`zeros-like`$(a)$
compute $\hat{s}$, Gaussian representation of $s$
compute $\hat{g}$ Gaussian representation of subgoal $g$
find $id_g$, effective gates indices for which $\hat{g} > 0.1$
compute net input matrix for $id_g$, $net = w^{(1)}_{id_g}\hat{s}$
**for all** $i$ in $id_g$ **do**
  compute $net^i_{kWTA} \leftarrow$`kWTA`$(net^i, k)$
  compute activity $h^i \leftarrow$ `sigmoid`$(net^i_{kWTA})$
  compute $q^i = w^{(2)}_i h^i$
  $q^{output} \leftarrow q^{output} + q^i$
**end for**
**return** $q^{output}$

**Backpropagation.** Updating $w$ given TD error $\delta$.

**for all** $i$ in $id_g$ **do**
  compute propagated error for hidden units $\delta^i_j \leftarrow \delta w^{(2)}_{i,a} \odot h^i \odot (1 - h^i)$
  $w^{(2)}_{i,a} \leftarrow w^{(2)}_a + \alpha\delta h^i$
  $w^{(1)}_i \leftarrow w^{(1)}_i + \alpha\hat{s}\delta^i_j$
**end for**

### 4.6.2 Intrinsic Motivation Performance Results

We tested the performance of our approach in the context of a dynamic rooms environment, with the agent rewarded for solving the key-car grid world task. We used the four following specific tasks to test the learning performance.

The agent was able to move in a two-dimensional grid world environment, containing one key and one car. The agent's location was bounded to be within a 2D Cartesian space of size $[0, 1] \times [0, 1]$. In the full task, as previously described, the

agent received the most reward if it first moved to the key and then moved to the car. In this version of the task, the locations of the key and the car are randomly selected, changing dynamically across training episodes. The agent received no reward or punishment for exploring the space, with the exception of a reward of $r = -2$ if the agent bumped into a wall. Positive rewards were different for different variants of the general task. The agent received complete sensory information of the entire environment (rather than just its own location), including the relative location of the key, the car, and the relative location of the agent, itself in the room. This additional information was used by the oracle to select the subgoals. Initially, the agent had no semantic knowledge about the objects in the environment. For example, it did not know that, in order to reach the car, it must grab the key first. This situation is illustrated in Figure 4.7(b). The agent's internal controller used $q(s, g, a; w)$ to select an action, $a_t$, based on the $\epsilon$-greedy policy. When tested, however, no exploration was allowed, so the policy for a given $g$ can be obtained as $\pi_{ag}(s, g) = \arg\max_a q(s, g, a; w)$. At regular intervals during training, we tested the ability of the controller to reach the key, and the car locations in four tasks.

- **Key Task, Hard Placement.** In this simplified version of the task, the agent was trained to move to the key, producing a policy, $\pi_{ag}$, for reaching a randomly located goal $g$ (key). This is illustrated in Figure 4.7(a). For each starting $s \in S$, a random goal, $g$, was assigned and the cumulative reward was obtained. We report the average reward scores and the average success percentage in Figure 4.8 (a) and (b), respectively.

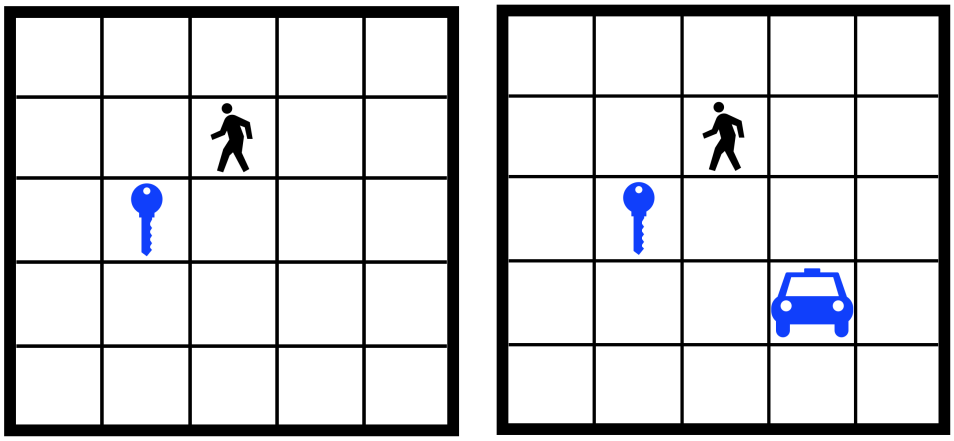
- **Key Task, Easy Placement.** This version of the task is the same as the last, except that the goal, $g$, was always randomly placed in a location adjacent to the starting state, $s$. (See Figure 4.7 (a).) We report the average reward scores and the average success percentage in Figure 4.8 (c) and (d), respectively.

- **Key-Car Task, Hard Placement.** In this version of the task, both the key, $g_{key}$, and the car, $g_{car}$, were randomly placed. The agent received positive reward when the agent moved to the key (+10) and subsequently moved to the car (+100). (See Figure 4.7 (b).) We report the average scores and the average success percentage in Figure 4.9 (a) and (b), respectively.

- **Key-Car Task, Easy Placement.** This version of the task is the same as the last, except that the key was always located at $(0, 0)$, and the car was always located at $(1, 1)$. We report the average reward scores and the average success percentage in Figure 4.9 (c) and (d), respectively.

### 4.6.3 Reusing Learned Skills

*Spacing* the state space, $\mathcal{S}$, through intrinsic motivation enables efficient learning for hierarchical tasks. Consider the rooms task shown in Figure 4.10(a). For each (slow



(a) (b)

Figure 4.7: In general, the agent received $r = +10$ reward for moving to the key and $r = +100$ if it then moved to the car. On each time step, the agent had four action choices $\mathcal{A} = \{$`North, South, East, West`$\}$. Bumping to the wall produced a reward of $r = -2$. There was no other reward or punishment from the environment for exploring the space. (a) Key task: agent needs to reach to the location of key. (b) Key-Car task: agent should first reach to the key and then to the car.

scale) time step, the meta-controller assigns a subgoal $g \in \{$`doorways, key, car`$\}$ to the controller. The trained policy can be used to achieve each subgoal, in turn. (See Figure 4.10 (b) to (f).) The controller is trained to solve the navigation task (in a single room gridworld) for any given goal $g$. The intrinsic motivation makes solving the rooms task easier since the agent can adapt the policy to the given goal $g$. It is important to note that we made an assumption that an oracle in the meta-controller provides the transformation from the state space to the input state for the network. This assumption is discarded in later experiments in Section 4.9

## 4.7 Unsupervised Subgoal Discovery

The performance of the meta-controller/controller framework depends critically on selecting good candidate subgoals for the meta-controller to consider.

What is a subgoal? In our framework, a subgoal is a state, or a set of related states, that satisfies at least one of these conditions:

1. It is close (in terms of actions) to a rewarding state. For example, in the rooms task in Figure 4.12(a), the key and lock are rewarding states.

2. It represents a set of states, at least some of which tend to be along a state transition path to a rewarding state.

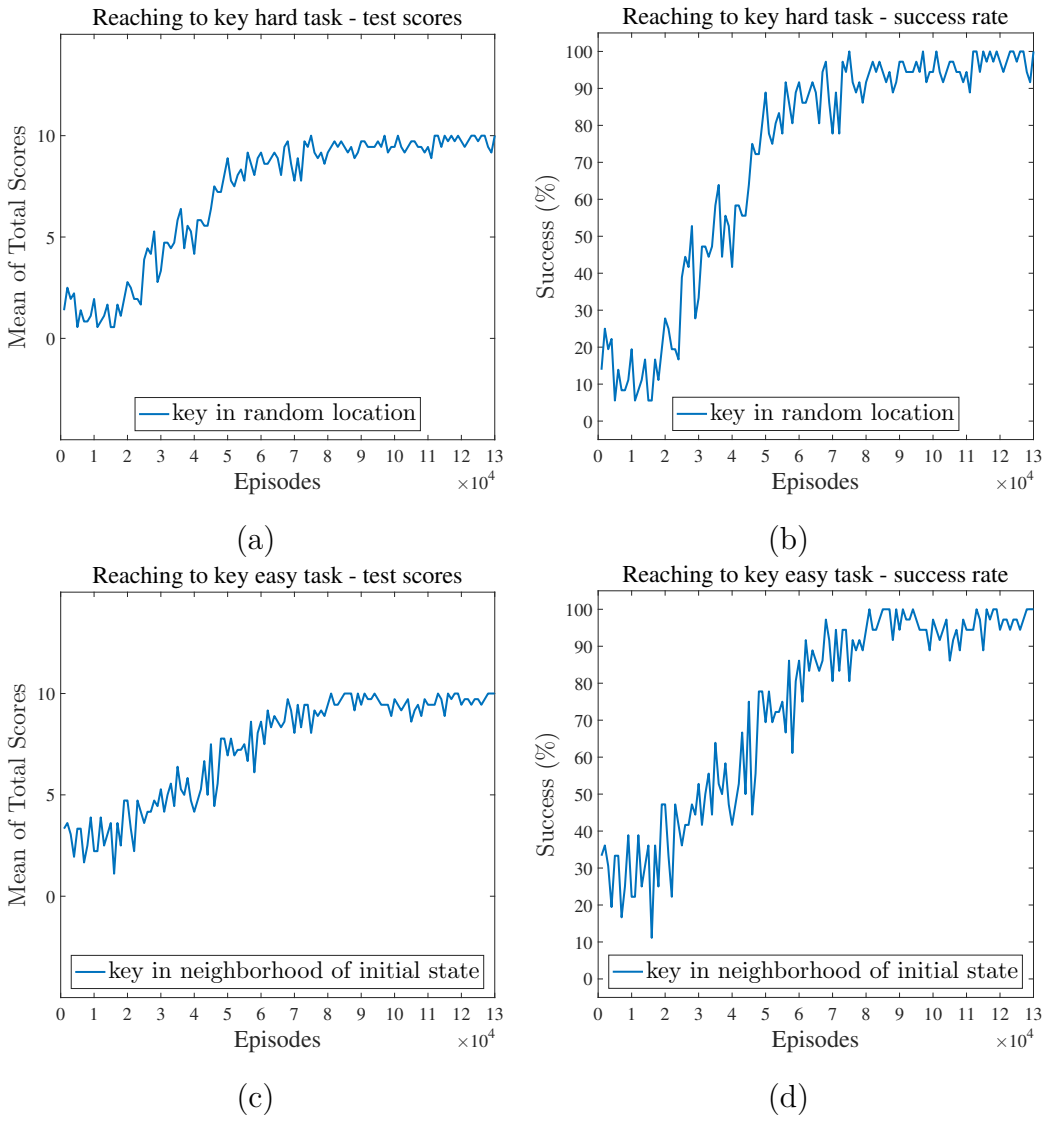



Figure 4.8: The test results for the task of moving to the key. Top: The key is located in a random location. Bottom: The key is randomly located in the neighborhood of the initial state. The *total scores* are the average of the total reward scores from all possible initial states. The *success rate* is the percentage of the test episodes in which the agent moves to the key.

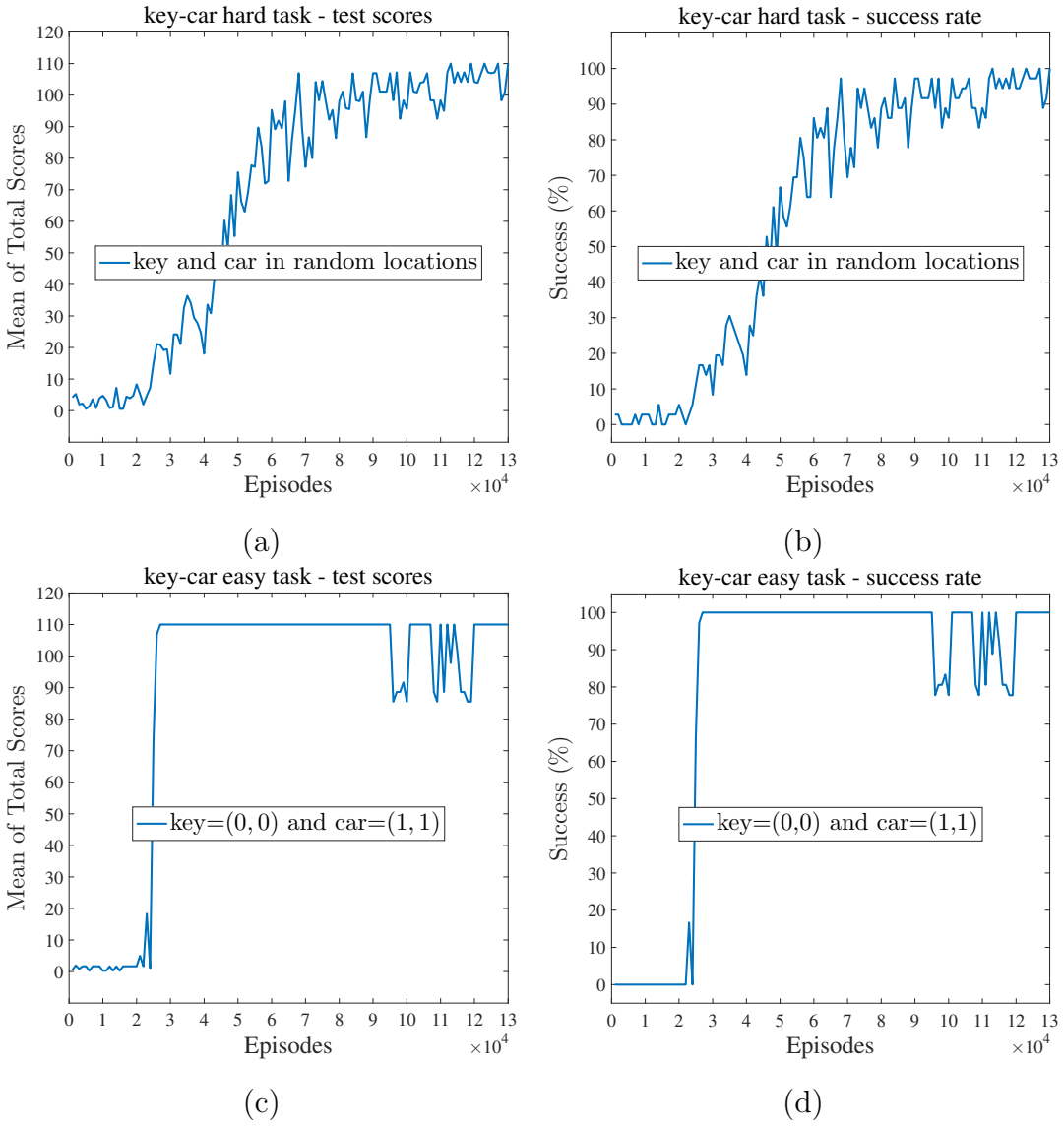



Figure 4.9: The test results for *key-car* task. Top: *hard placement* — the key and the car are placed in random locations. Bottom: *easy placement* — the key is located at (0,0) and the car is located at (1,1). The *total scores* are the average of the total scores form all possible initial states. The *success rate* is the percentage of the test episodes in which the agent successfully moves to the key and then to the car.

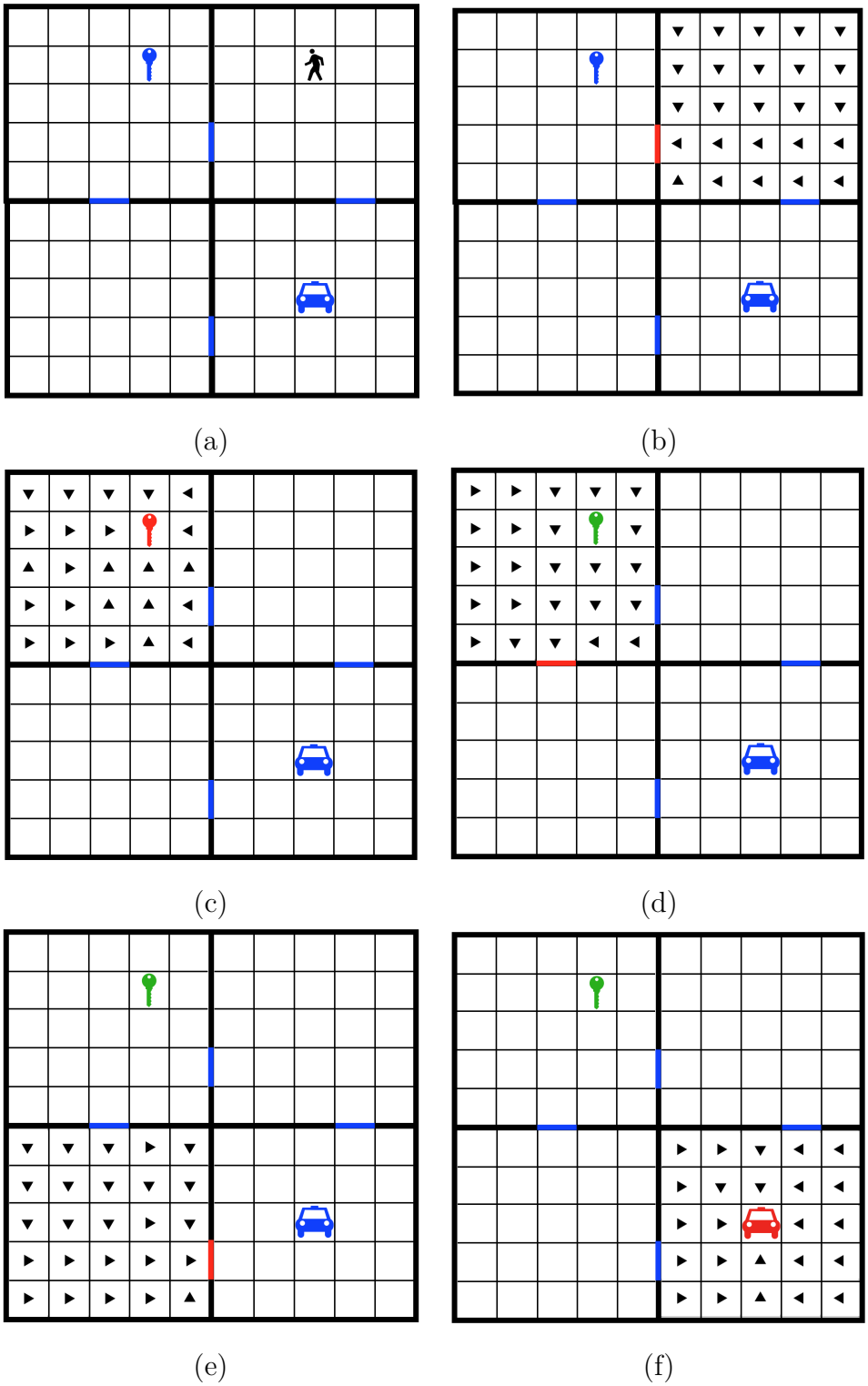



Figure 4.10: Reusing the navigation skill to solve the rooms task. At each time step, an oracle selected a subgoal for the agent (red locations). The agent with the pretrained navigation skill successfully accomplishes all of the subgoals assigned by the oracle. (a) The starting configuration. (b) Subgoal: doorway between room 1 and 2. (c) Subgoal: moving to the key. (d) Subgoal: doorway between room 2 and 3. (e) Subgoal: doorway between room 3 and 4. (f) Subgoal: moving to the car.

For example, in the rooms task, the red room, as illustrated in Figure 4.12 (a), should be visited to move from the purple room to the blue room in order to pick up the key. Thus any state in the red room is a reasonably good subgoal for an agent currently in the purple room. Similarly, the states in the blue room are all reasonably good subgoals for an agent currently in the red room. The doorways between rooms can also be considered as good subgoals, since entering these states allows for the transition to a set of states that may be closer to rewarding states.

Our strategy involves leveraging the set of recent transition experiences that must be recorded for value function learning, regardless. Unsupervised learning methods applied to sets of experiences can be used to identify sets of states that may be good subgoal candidates. We focus specifically on two kinds of analysis that can be performed on the set of transition experiences. We hypothesize that good subgoals might be found by (1) attending to the states associated with *anomalous* transition experiences and (2) clustering experiences based on a similarity measure and collecting the set of associated states into a potential subgoal. Thus, our proposed method merges *anomaly (outlier) detection* with the $K$-means clustering of experiences. The unsupervised subgoal discovery method is summarized in Algorithm 10.

**Algorithm 10** Unsupervised Subgoal Discovery Algorithm

```
for each e = (s, a, r, s') stored in D do
    if experience e is an outlier (anomaly) then
        Store s' to the subgoals set G
        Remove e from D
    end if
end for
Fit a K-means Clustering Algorithm on D using previous centroids as initial points
Store the updated centroids to the subgoals set G
```

### 4.7.1 Anomaly Detection

The anomaly (outlier) detection process identifies states associated with experiences that differ significantly from the others. In the context of subgoal discovery, a relevant anomalous experience would be one that includes a substantial positive reward in an environment in which reward is sparse. We propose that the states associated with these experiences make for good candidate subgoals. For example, in the rooms task, transitions that arrive at the key or the lock are quite dissimilar to most transitions, due to the large positive reward that is received at that point.

Since the goal of RL is maximizing accumulated (discounted) reward, these anomalous experiences, involving large rewards, are ideal as subgoal candidates. (Experiences involving large negative rewards are also anomalous, but make for poor subgoals. As long as these sorts of anomalies do not greatly outnumber others, we expect that the meta-controller can efficiently learn to avoid poor subgoal choices.) Large changes

in state features can also be marked as anomalous. In some computer games, like *Montezuma's Revenge*, each screen represents a room, and the screen changes quickly when the agent moves from one room to another. This produces a large distance between two consecutive states. Such a transition can be recognized simply by the large instantaneous change in state features, marking the associated states as reasonable candidate subgoals. There is a large literature on anomaly detection (Hodge and Austin, 2004), in general, offering methods for applying this insight. Heuristic meta-parameter thresholds can be used to identify dissimilarities that warrant special attention, or unsupervised machine learning methods can be used to model the joint probability distribution of state variables, with low probability states seen as anomalous.

### 4.7.2 K-Means Clustering

The idea behind using a clustering algorithm is "spatial" state space abstraction and dimensionality reduction with regard to the internal representations of states. If a collection of transition experiences are very similar to each other, this might suggest that the associated states are all roughly equally good as subgoals. Thus, rather than considering all of those states, the learning process might be made faster by considering a representative state (or smaller set of states), such as the centroid of a cluster, as a subgoal. Furthermore, using a simple clustering technique like $K$-means clustering to find a small number of centroids in the space of experiences is likely to produce centroid subgoals that are dissimilar from each other. Since rewards are sparse, this dissimilarity will be dominated by state features. For example, in the rooms task, the centroids of $K$-means clusters, with $K = 4$, lie close to the geometric centers of the rooms, with the states within each room coming to belong to the corresponding subgoal's cluster. In this way, the clustering of transition experiences can approximately produce a coarser representation of state space, in this case replacing the fine grained "grid square location" with the coarser "room location".

### 4.7.3 Mathematical Interpretation

The value of a state, $V_\pi(s)$, is defined as the expected future rewards, following a policy $\pi$

$$V_\pi(s) \triangleq \mathbb{E}\Big[\sum_{t=0}^{\mathcal{T}} \gamma^t r_t | s, \pi\Big], \tag{4.9}$$

where $\mathcal{T}$ is a termination time, and $\gamma < 1$. In the model-free HRL framework, $V_\pi(s)$ can be approximated by a sequence of values of the meta-controller's value function for one subgoal after another

$$V(s) \approx Q(s, g_1) + \gamma^{T_1} Q(g_1, g_2) + \gamma^{T_1+T_2} Q(g_2, g_3) + \dots \tag{4.10}$$

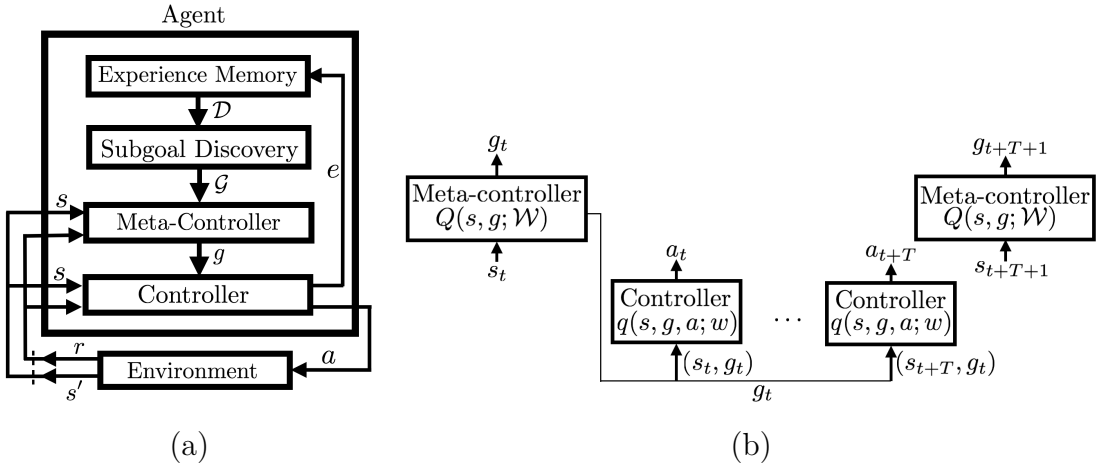
where $T_i \leq T$ is effective number of controller steps to accomplish subgoal $g_i$ (note that $\mathcal{T} = T_1 + T_2 + \dots$). We assume that the controller has learned a good policy through the process of intrinsic motivation learning. Since the rewards are sparse, the value of any state close to an anomalous subgoal is roughly equal to the immediate reward $r$ (since the future rewards vanish for a small discount factor $\gamma$). The $K$-means clustering algorithm partitions the state space into $K$ regions. The value of states in a cluster close to an anomalous subgoal $g$ is approximately $\gamma^{T_1} r$, since it takes $T_1$ steps from states in this region to arrive the rewarding state, and obtain the reward $r$. The clustering algorithm takes into consideration the distance between experiences that do not contain anomalous ones. Therefore, the states in a cluster have similar values, because, for each state in a cluster, it takes approximately the same number of steps to reach a rewarding state (an anomalous subgoal).

## 4.8 A Unified Model-Free HRL Framework

In this section, we introduce a unified method for model-free HRL, so that all three HRL subproblems can be solved jointly. Our intuition, shared with other researchers, is that hierarchies of abstraction will be critical for successfully solving problems with sparse delayed feedback. To be successful, the agent should represent knowledge at multiple levels of spatial and temporal abstraction. Appropriate abstraction can be had by identifying a relatively small set of states that are likely to be useful as *subgoals* and jointly learning the corresponding skills of achieving these subgoals, using intrinsic motivation.

Inspired by Kulkarni et al. (2016), we start by using two levels of hierarchy (Figure 4.11). The more abstract level of this hierarchy is managed by a *meta-controller* which guides the action selection processes of the lower level *controller*. Separate value functions are learned for the meta-controller and the controller as shown in Figure 4.11(b).

The conceptual components of HRL — temporal abstraction, intrinsic motivation, and unsupervised subgoal discovery — can be unified into a single model-free HRL framework. The major components of this framework, and the information flow between these components, are schematically displayed in Figure 4.11 (a). At time $t$, the meta-controller observes the state, $s = s_t$, from the environment and chooses a subgoal, $g = g_t$, either from the discovered subgoals or from a random set of states (to promote exploration). The controller receives an input tuple, $(s, g)$, and is expected to learn to implement a subpolicy, $\pi(a|s, g)$, that solves the *subtask* of reaching from $s$ to $g$. The controller selects an action, $a$, based on its policy, in our case directly derived from its value function, $q(s, g, a; w)$. After one step, the environment updates the state to $s'$ and sends a reward $r$. The agent's experience $(s, a, s', r)$ is stored in the experience memory, $\mathcal{D}$. The intrinsic transition experience $(s, g, a, \tilde{r}, s')$ is stored in the controller's experience memory, $\mathcal{D}_1$. If the internal critic detects that the resulting state, $s'$, is the current goal, $g$, the experience $(s_t, g, G, s_{t'})$ is stored in the meta-controller experience memory, $\mathcal{D}_2$, where $s_t$ is the state that prompted the se-




(a) (b)

Figure 4.11: (a) The information flow in the unified Model-Free Hierarchical Reinforcement Learning Framework. (b) Temporal abstraction in the meta-controller/controller framework.

lection of the current subgoal, and $s_{t'} = s_{t+T}$ is the state when the meta-controller assigns the next subgoal, $g' = g_{t'}$. The experience memory sets are typically used to train the value function approximators for the meta-controller and the controller by sampling a random minibatch of recent experiences. The unsupervised subgoal discovery mechanism exploits the underlying structure in the experience memory $\mathcal{D}$ using unsupervised anomaly detection and $K$-means clustering. A detailed description of the unified representation learning in our model-free HRL framework is outlined in Algorithm 11.

## 4.9 Experiments on Unified HRL Framework

We conducted simulation experiments in order to investigate the ability of our unsupervised subgoal discovery method to discover useful subgoals, as well as the efficiency of our unified model-free hierarchical reinforcement learning framework. The simulations were conducted in two environments with sparse delayed feedback: a variant of the rooms task, shown in Figure 4.12(a), and the "Montezuma's Revenge" game, shown in Figure 4.17(a).

All code was implemented in the Python using Pytorch, NumPy, Opencv, and SciPy libraries and is available at `https://github.com/root-master/unified-hrl`.

### 4.9.1 4-Room Task with Key and Lock

Consider the task of navigation in the *4-room environment with a key and a lock*, as shown in Figure 4.12(a). This is the same task that was explored in earlier parts of this chapter. While this task was inspired by the *rooms* environment introduced by Sutton, et al. (1999), it is much more complex. As before, the agent not only needs

**Algorithm 11** Unified Model-Free HRL Algorithm

Pretrain controller using Algorithm 8 on a set of random subgoals $\mathcal{G}'$
Initialize experience memories $\mathcal{D}$, $\mathcal{D}_1$ and $\mathcal{D}_2$
Walk controller for $M'$ episodes on random subgoals $\mathcal{G}'$, and store $(s, a, s', r)$ to $\mathcal{D}$
Run Unsupervised Subgoal Discovery on $\mathcal{D}$ to initialize $\mathcal{G}$
**for** episode $= 1, \ldots, M$ **do**
  Initialize state $s_0 \in \mathcal{S}$, $s \leftarrow s_0$
  $G \leftarrow 0$
  $g \leftarrow$`EPSILON-GREEDY`$(Q(s, \mathcal{G}; \mathcal{W}), \epsilon_2)$
  **repeat** for each step $t = 1, \ldots, T$
    compute $q(s, g, a; w)$
    $a \leftarrow$`EPSILON-GREEDY`$(q(s, g, \mathcal{A}; w), \epsilon_1)$
    Take action $a$, observe $s'$ and external reward $r$
    Compute intrinsic reward $\tilde{r}$ from internal critic
    Store controller's intrinsic experience, $(s, g, a, \tilde{r}, s')$ to $\mathcal{D}_1$
    Store agent's transition experience, $(s, a, r, s')$ to $\mathcal{D}$
    Sample $J_1 \subset \mathcal{D}_1$ and compute $\nabla L$
    Update controller's parameters, $\quad w \leftarrow w - \alpha_1 \nabla L$
    Sample $J_2 \subset \mathcal{D}_2$ and compute $\nabla \mathcal{L}$
    Update meta-controller's parameters, $\quad \mathcal{W} \leftarrow \mathcal{W} - \alpha_2 \nabla \mathcal{L}$
    $s \leftarrow s', \quad G \leftarrow G + r$
    Decay exploration rate of controller $\epsilon_1$
    **if** experience $e$ is an outlier (anomaly) **then**
      Store $s'$ to the subgoals set $\mathcal{G}$
      Remove $e$ from $\mathcal{D}$
    **end if**
  **until** $s$ is terminal or subgoal $g$ is attained
  Decay exploration rate of meta-controller $\epsilon_2$
  Store meta-controller's experience, $(s_0, g, G, s')$ to $\mathcal{D}_2$
  Fit a $K$-means clustering on $\mathcal{D}$ every $N$ step to update centroids of $\mathcal{G}$
**end for**

to learn how to navigate form any arbitrary state to any other state, but it also needs to visit some states in a specific temporal order. At the beginning of each episode, the agent is initialized in an arbitrary location in an arbitrary room. The agent has four possible move actions, $\mathcal{A} = \{North, South, East, West\}$, on each time step. The agent receives $r = +10$ reward for reaching the key and $r = +40$ if it moves to the lock while carrying the key (i.e., any time after visiting the key location during the same episode). Bumping into a wall boundary is punished with a reward of $r = -2$. There is no reward for just exploring the space. Learning in this environment with sparse delayed feedback is challenging for a reinforcement learning agent. To successfully solve the task, the agent should represent knowledge at multiple levels of spatial and temporal abstractions. The agent should also learn to explore the environment

efficiently.

We first examined the unsupervised subgoal discovery algorithm over the course of a random walk. The agent was allowed to explore the *4-room* environment for 100 episodes. Each episode ended either when the task was completed or after reaching a maximum time step limit of 200. The agent's experiences, $e = (s, a, r, s')$, were collected in an experience memory, $\mathcal{D}$. The stream of external rewards for each transition was used to detect *anomalous* subgoals (Figure 4.13(a)). We applied a heuristic anomaly detection method for the streaming rewards that was able to differentiate between the rare large positive rewards and the regular small ones. These peaks, as shown in Figure 4.13(a), corresponded to the experiences in which the key was reached ($r = +10$) or the experience of reaching the lock after obtaining the key.

We also applied a $K$-means clustering algorithm to the experience memory. (See Algorithm 11.) The centroids of the $K$-means clusters (with $K = 4$) are plotted in Figure 4.12(b). The clusters, along with the anomalous states, were collected into the set of subgoals. By choosing $K = 4$ for the number of clusters (or centroids), the discovered centroids roughly correspond to the centers of the rooms, and the clusters correspond to the rooms. But, the choice of $K = 4$ comes from our knowledge of the spatial structure of this environment. Here, we show that other choices for $K$ lead to different, but still useful, clusterings of the state space. Indeed, all we expect from the clustering algorithm in unsupervised subgoal discovery is to divide the state space into clusters of states with roughly similar values, and the choice of $K$ is not crucial. Also, any good spatial clustering method would work equally well. For example, with the number of clusters, $K = 6$, we saw a clustering in which, two of the four rooms were divided into two clusters (see Figure 4.12(c)). We repeated this experiment for $K = 8$, where we saw equally useful clusters, with two room containing two cluster centroids, one room containing three clusters, and one room only with one cluster (see Figure 4.12(d)). The K-means algorithm in our unsupervised subgoal discovery can be incremental, using the previous centroids as initial points for the next iteration. Therefore, the configuration of clusters were different over training, but the results of clustering were useful regardless.

In summary, our unified model-free HRL framework (Algorithm 11) does not rely on a particular choice of $K$. To prove this claim, we trained the agent in the unified model-free HRL framework for different numbers of clusters, $K = 4$, $K = 6$, and $K = 8$.

In these simulations, learning occurred in one unified phase consisting of 100,000 episodes. The meta-controller and the controller, and unsupervised subgoal discovery, were trained all together. (See Algorithm 11.) Value function approximators were implemented as multi-layer artificial neural networks as shown in Figure 4.14. The controller network, $q(s, g, a; w)$, took the state, $s$, and the goal, $g$, as inputs. States were presented to the network as Cartesian coordinates, with separate pools of inputs for each of the two dimensions. The subgoal was initially chosen randomly from the set of discovered subgoals, resulting from unsupervised subgoal discovery during early random walks of the agent. The meta-controller value function receives a one-

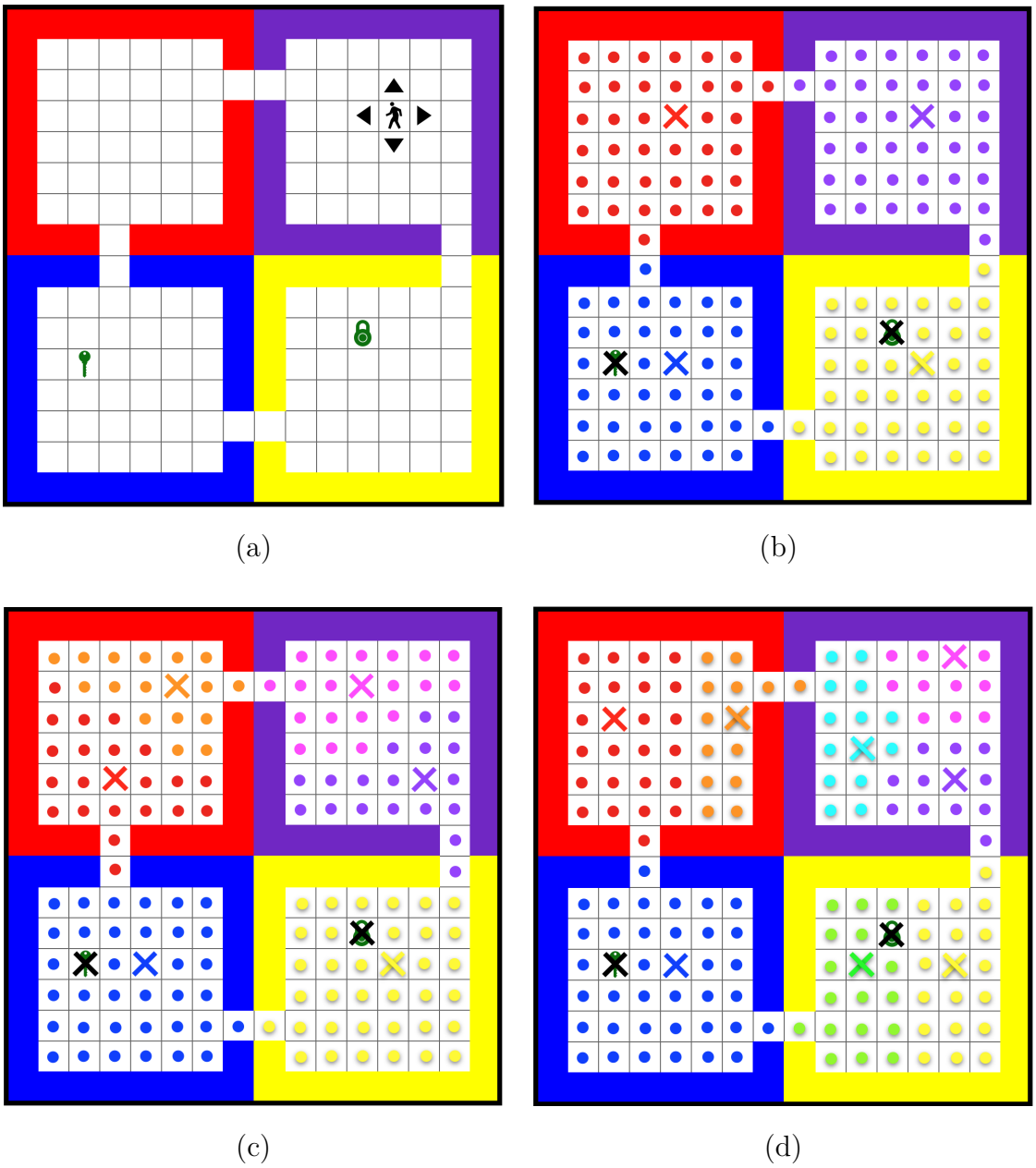


Figure 4.12: (a) The *4-room* task with a key and a lock. (b) The results of the unsupervised subgoal discovery algorithm with *anomalies* marked with black Xs and *centroids* with colored ones. The number of clusters in $K$-means algorithm was set to $K = 4$. (c) The result of the unsupervised subgoal discovery for $K = 6$. (d) The results of the unsupervised subgoal discovery for $K = 8$.

hot encoding of the current state, computed by converting the current state to the index of the corresponding subgoal. The meta-controller outputs a one-hot encoding of the best subgoal. The controller receives a Gaussian-blurred representation of current state variables (Cartesian coordinates) gated by the current subgoal, and it

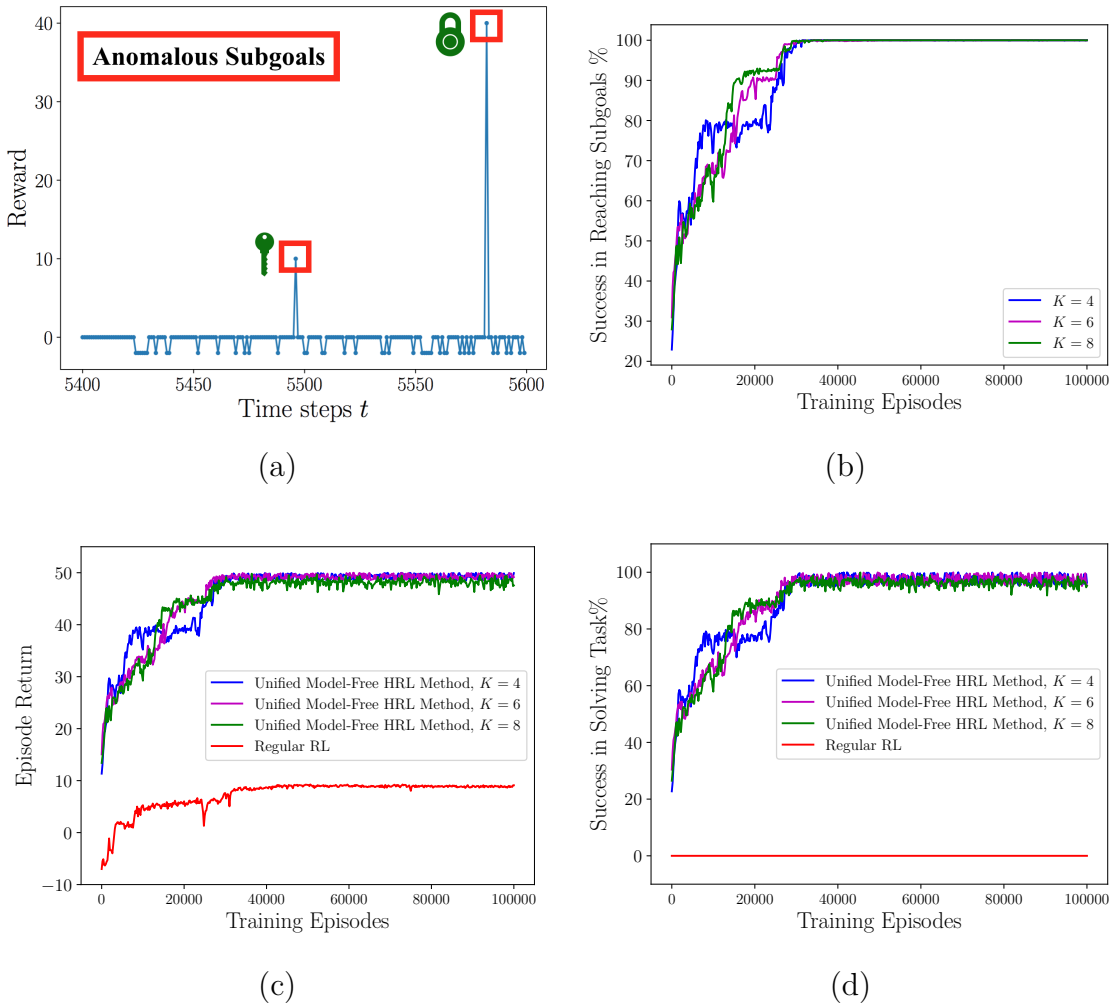



Figure 4.13: (a) Reward over an episode, with anomalous points corresponding to the key ($r = +10$) and the lock ($r = +40$). (b) The average success of the controller in reaching subgoals over 200 consecutive episodes. (c) The average episode return. (d) The average success rate for solving the 4-room task.

produces a sparse conjunctive encoding over hidden units using a $k$-Winners-Take-All mechanism, akin to lateral inhibition in cortex (Rafati and Noelle, 2015; O'Reilly and Munakata, 2001). This is then mapped onto the controller value function output for each possible action. Most previously published subgoal discovery methods focus on finding the doorways (funnel type subgoals) (Goel and Huber, 2003; Simsek et al., 2005). With $K = 4$, the doorways can be discovered as boundaries between adjacent clusters. Note that our method is not strongly task dependent, so the choice of $K$ is not crucial to the learning of meaningful representations. The results of clustering for different values of $K$ are shown in Figure 4.12 (b-d).

When a centroid was selected as a subgoal, if the agent entered any state in the corresponding cluster, the subgoal was considered attained. Thus, the controller

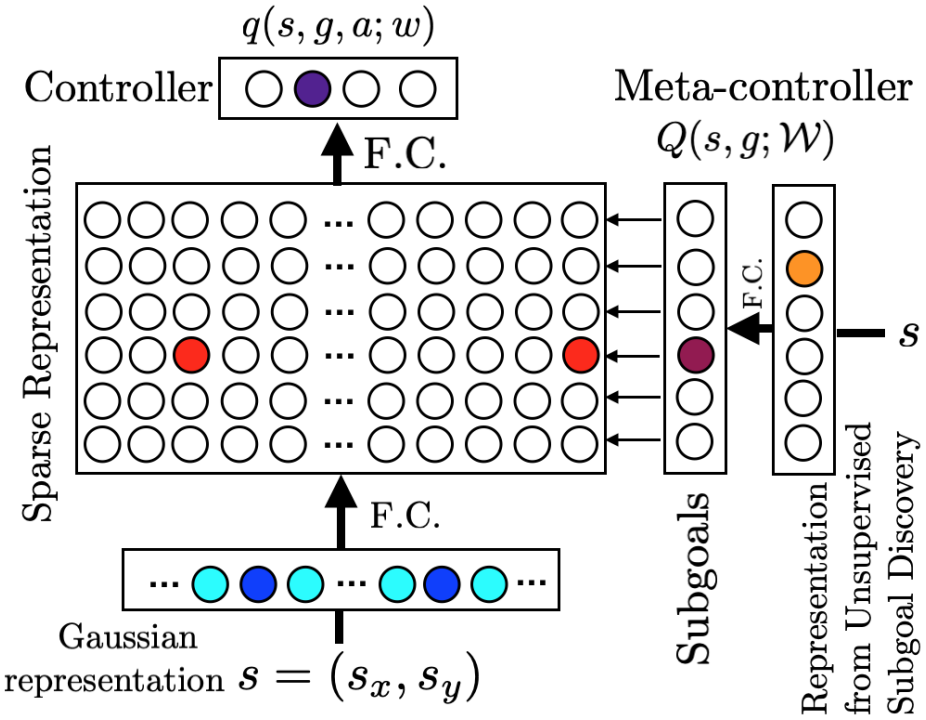



Figure 4.14: Integrated meta-controller and controller network architecture.

essentially learned how to navigate from any location to any state cluster and also to any of the anomalous subgoals (key and door). The learning rate was $\alpha = 0.001$, the discount factor was $\gamma = 0.99$, and the exploration rate was set to $\epsilon_1 = \epsilon_2 = 0.2$. The average success rate of the controller in achieving subgoals is shown in Figure 4.13(b).

The average return, over 200 consecutive episodes, is shown in Figure 4.13(c). The agent very quickly converged on the optimal policies and collected the maximum reward (+50). The high exploration rate of 0.2 caused high stochasticity, but the meta-controller and controller could robustly solve the task on more than 90% of the episodes very early in training. After about 40,000 episodes, the success rate was 100%, as shown in Figure 4.13(d). There was no significant difference in the results of learning for different choices of the number of clusters, $K$.

We compared the learning efficiency of our unified HRL method with the performance resulting from training a value approximation network with a regular, non-hierarchical, RL algorithm, TD SARSA (Sutton and Barto, 2017). The function approximator that we used for $Q(s, a; w)$ matched that of the controller, equating for computational resources, and we used the same values for the training hyper-parameters. The regular RL agent could only reach the key before becoming stuck in that region, due to the high local reward. Despite the very high exploration rate used, the regular RL agent was not motivated to explore the rest of the state space to reach the lock and solve the task. Results are shown in Figure 4.13(c) and (d) (red

lines).

In our unified model-free HRL, the intrinsic motivation learning provides an efficient exploration policy for successful subgoal discovery using the unsupervised subgoal discovery algorithm. We examined the role of intrinsic motivation learning in an efficient exploration of the state space by computing the *rate of coverage*, i.e. the number of visited states divided by the size of the state space, calculated during training. The results for the rate of coverage, using different methods, are shown in Figure 4.15. A random walk of the agent could only cover 30% of the states (grid square) after 1,000 episodes, with each episode consisting of 32 steps. When the agent was trained by a regular Q-learning method, it could only learn a sub-optimal policy and reach the *key*, but it didn't have a motivation to explore the other regions of the state space and find the other rewarding states, i.e. the *lock*. The rate of coverage using the regular RL method was 40% after 1,000 episodes of training. Intrinsic motivation learning, coupled with the unsupervised subgoal discovery algorithm and a random meta-controller, visited 67% of the states after 1,000 episodes. Our unified model-free HRL method, in which the intrinsic motivation learning was integrated with the unsupervised subgoal discovery and meta-controller learning, could successfully cover 100% of the state space and discover all the useful subgoals in less than 1,000 episodes. This led to successfully solving the task (see Figure 4.13(d)).

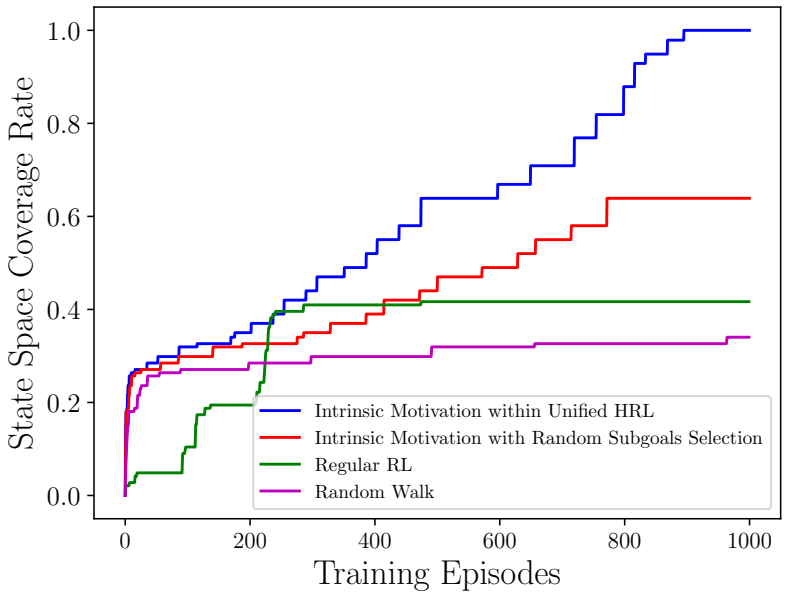



Figure 4.15: The rate of coverage in the rooms task. Plotted is the number of visited states as a fraction of the total size of the state space.

It is worth noting that this task involves a *partially observable Markov decision process* (POMDP), because information about whether or not the agent has the key is not visible in the state. This hidden state information poses a serious problem for standard RL algorithms, but our HRL agent was able to overcome this obstacle.

Through meta-controller learning, the hidden information became implicit in the selected subgoal, with the meta-controller changing the current subgoal once the key is obtained. In this way, our HRL framework is able to succeed in task environments that are effectively outside of the scope of standard RL approaches.

### 4.9.2 Montezuma's Revenge

We applied our HRL approach to the first room of the game *Montezuma's Revenge*. (See Figure 4.17(a).) The game is well-known as a challenge for RL agents because it requires solving many subtasks while avoiding traps. Having only sparse delayed reward feedback to drive learning makes this RL problem extremely difficult. The agent should learn to navigate the *man* in red to the *key* by: (1) climbing the *middle ladder* (2) climbing the *bottom right ladder* (3) climbing the *bottom left ladder* (4) moving to the key. After picking up the key ($r = +100$), the agent should return back, reversing the previous action sequence, and attempt to reach the *door* ($r = +300$) and exit the room. The moving skull at the bottom of the screen, which ends an episode upon contact, makes obtaining the key extremely difficult. The episode also ends unsuccessfully if the man falls off of a platform.

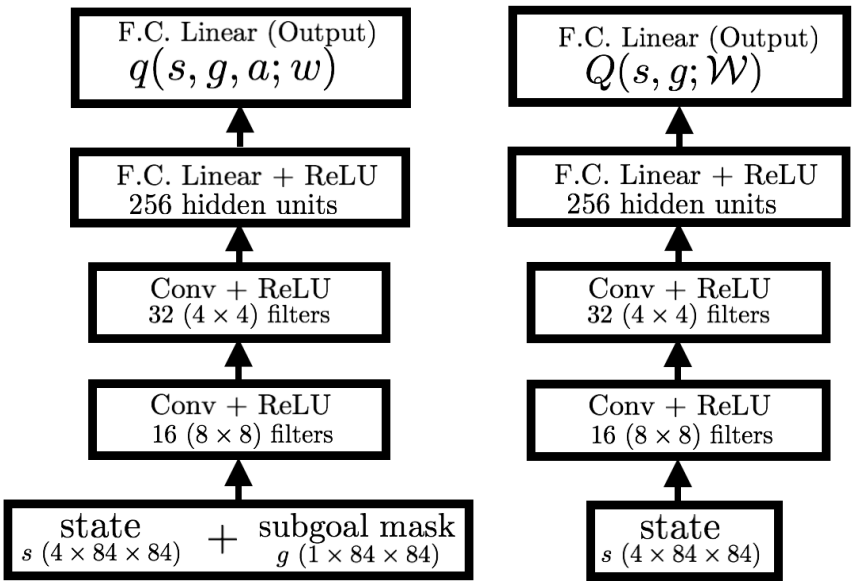
DeepMind's Deep Q-Learning (DQN) algorithm (Mnih et al., 2015), which surpassed human performance on many ATARI 2600 games, failed to learn this game since the agent did not reach any rewarding state during exploration.

In this problem, the agent requires the skills arising from intrinsic motivation learning in order to explore the environment in a more efficient way (Kulkarni et al., 2016). Our HRL approach supports the learning of such skills. The meta-controller and the controller were trained in two phases.

In Phase I (pretraining), the controller was trained to move the man from any location in the given frame, $s$, to any other location specified in a subgoal frame, $g$. Unsupervised object detection using computer vision algorithms can be challenging (Kulkarni et al., 2016; Fragkiadaki et al., 2015). We made the simplifying assumption that, in many games, edges were suggestive of objects, and the locations of objects made for good initial subgoals. An initial set of "interesting" subgoal locations were identified using a custom edge detection algorithm, avoiding empty regions as subgoals. We used the Canny edge detection algorithm on one image of the game, and chose 20 random locations, as initial subgoals, from the identified edges (see Figure 4.17(b)). These locations were used in Phase I of training to train the controller through intrinsic motivation, using Algorithm 8. Note that edge detection was only performed to identify Phase I subgoals. Specifically, it was *not* used to change or augment the state representation in any way.

We used a variant of the DQN deep Convolutional Neural Network (CNN) architecture for approximation of the controller's value function, $q(s, g, a; w)$ (see Figure 4.16 (a)). The input to the controller network consisted of four consecutive frames of size $84 \times 84$, encoding the state, $s$, and an additional frame binary mask encoding the subgoal, $g$. The concatenated state and subgoal frames were passed to the network,

and the controller then selected one of 18 different joystick actions based on a policy derived from $q(s, g, a; w)$.




(a) (b)

Figure 4.16: (a) The CNN architecture for the controller's value function. (b) The CNN architecture for the meta-controller's value function.

During intrinsic motivation learning, the recent experiences were saved in an experience memory, $\mathcal{D}$, with a size of $10^6$. In order to support comparison to previously published results, we used the same learning parameters as DeepMind's DQN (Mnih et al., 2015). Specifically, the learning rate, $\alpha$, was set to to be 0.00025, with a discount rate of $\gamma = 0.99$. During Phase I learning, we trained the network for a total of $2.5 \times 10^6$ time steps. The exploration probability parameter, $\epsilon_1$, decreased from 1.0 to 0.1 in the first million steps and remained fixed after that.

After every $100,000$ time steps, we applied our unsupervised subgoal discovery method to the contents of the experience memory in order to find new subgoals, both anomalies and centroids, using $K$-means clustering with $K = 10$. As shown in Figure 4.17(c), the unsupervised learning algorithm managed to discover the location of the key and the doors in this way. It also identified useful objects such as ladders, platforms, and the rope. These subgoals were used to train the meta-controller and controller. The intrinsic motivation learning process played a major role in the successful subgoal discovery of subgoals, producing a policy that encouraged an efficient exploration of the state space. A random walk of the agent could only discover a small subset of the subgoals, e.g., two ladders and the rope (see Figure 4.17 (d)). But, the unsupervised subgoal discovery method used during the intrinsic motivation learning process could discover all of the useful subgoals (see Figure 4.17(c)).

In Phase II, we trained the meta-controller, the controller, and unsupervised subgoal discovery jointly together using Algorithm 11. We reset the exploration rates, $\epsilon_1$,

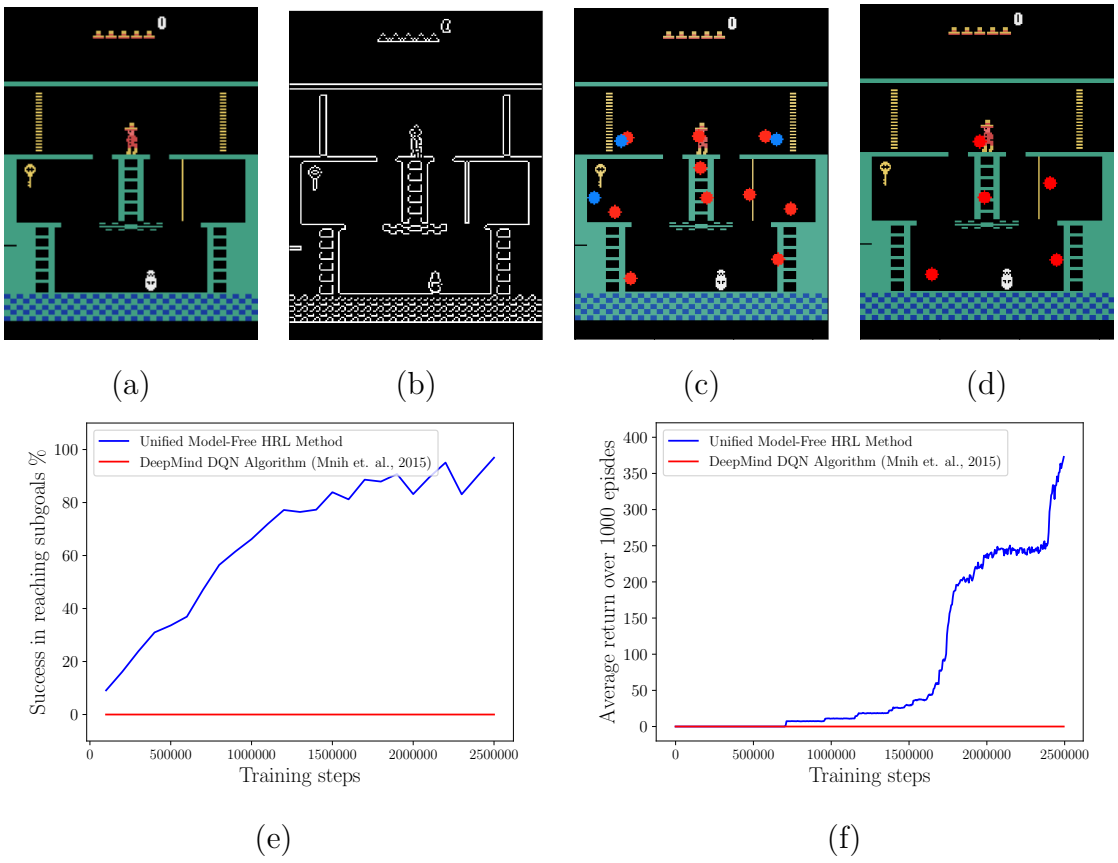



Figure 4.17: (a) The first screen of the Montezuma's Revenge game. (b) The results of the Canny edge detection algorithm on a single image of the game. (c) The results of the unsupervised subgoal discovery algorithm during the intrinsic motivation learning of the controller in the first room of the Montezuma's Revenge game. Blue circles are the anomalous subgoals and the red ones are the centroids of clusters. (d) The results of the unsupervised subgoal discovery for a random walk. (e) The success of the controller in reaching subgoals. (f) The average game score.

and $\epsilon_2$ to 1. The exploration probability parameters decreased from 1.0 to 0.1 in the first million steps and remained fixed after that. We ran the unsupervised subgoal discovery method every $100,000$ time steps to update the centroids of the clusters. We used an architecture based on the DQN CNN (Mnih et al., 2015), as shown in Figure 4.16(b), for the meta-controller's value function, $Q(s, g; \mathcal{W})$. All the rewarding (anomalous) subgoals were discovered in Phase I. We used the non-overlapping discovered subgoals, which resulted in a set of 11 subgoals, $\mathcal{G}$. At the beginning of each episode, the meta-controller assigned a subgoal, $g \in \mathcal{G}$, based on an epsilon-greedy policy derived from $Q(s, g; \mathcal{W})$. The controller then attempted to reach these subgoals. The controller's experience memory, $\mathcal{D}_1$, had a size of $10^6$, and the size of the meta-controller's experience memory, $\mathcal{D}_2$, was $5 \times 10^4$.

The cumulative rewards for the game episodes are shown in Figure 4.17(e). After

about 1.8 million time steps, the controller managed to reach the key subgoal more frequently. The average success of intrinsic motivation learning over $100,000$ consecutive episodes is depicted in Figure 4.17(f). After about 2 million learning steps, the meta-controller regularly chose the proper subgoals for collecting the maximum reward (+400).

## 4.10 Neural Correlates of Model-Free HRL

This work has been inspired, in part, by theories of reinforcement learning in the brain. These theories often involve interactions between the striatum and neocortex. The temporal difference learning algorithm, which is a model-free RL, account for the role of the midbrain dopaminergic system (Schultz et al., 1993). The actor-critic architectures for RL have drawn connections within the basal ganglia and cerebral cortex. The RL-based accounts have also addressed the learning processes for motor control, working memory, and habitual and goal-directed behaviors.

There is some evidence that temporal abstraction in HRL might map onto regions within the dorsolateral and orbital prefrontal cortex (PFC) (Botvinick et al., 2009), allowing the PFC to provide hierarchical representations to the basal ganglia.

More recent discoveries reveal a potential role for medial temporal lobe structures, including the hippocampus, in planning and spatial navigation (Botvinick and Weinstein, 2014), utilizing a hierarchical representation of space (Chalmers et al., 2016). There are evidences that hippocampus serve in model-based and model-free HRL with both flexibility and computational efficiency (Chalmers et al., 2016). Perhaps, the most salient aspects of the hippocampus is the existence of *place cells*. (Strange et al., 2014), which activate in particular regions of an environment. Place cells in the dorsal hippocampus represent small regions while those in the ventral hippocampus represent larger regions (Chalmers et al., 2016). The fact that the hippocampus learns representations at multiple scales of abstraction supports the idea that the hippocampus might be a major component of the subgoal discovery mechanism in the brain. For navigation in the 4-room task, we see that the clustering algorithm divides the state space into a few big regions (ventral hippocampus), and the anomaly detection algorithm detects much smaller rewarding regions (dorsal hippocampus).

There are also studies of interactions between the hippocampus and the PFC that are directly related to our unsupervised subgoal discovery method. Preston and Eichenbaum (2013) illustrated how novel memories (like *anomalous* subgoals) could be reinforced into permanent storage. Additionally, their studies suggest how PFC may be important for finding new meaningful representations from memory replay of experiences. This phenomena is similar to our clustering of experience memory.

## 4.11 Future Work

### 4.11.1 Learning Representations in Model-based HRL

As mentioned in Chapter 2, the difference between model-based RL, and model-free RL lies fundamentally in the type of information the agent stores in memory (Botvinick and Weinstein, 2014). In model-free RL, the agent stores a representation of the value function, which is an estimate of the cumulative future rewards the agent expects, when beginning in a particular state with a particular action. Updates to the representations of the value function are driven by the temporal difference error, which is a reward prediction error generated based on the agent's experience during direct interaction with the environment.

On the other hand, model-based RL, does not store a value function. Instead, the agent maintains an internal model of the environment, including a model of the reward function and a model of the state transition probabilities. The state transition model, $\hat{p}(s'|s,a)$, predicts a distribution over next states as an outcome of a particular action in a particular state. The reward model, $\hat{r}(s,a)$, represents an estimate of the immediate reward associated with individual states or actions. An RL agent equipped with these two knowledge structures can use planning to predict the outcome of future actions, before direct interaction with the environment. After executing a particular plan, the error of the agent's internal models can be used to update the models from the experience with the environment.

Although learning models of the environment adds to the computational complexity of representation learning, it is important to note that model learning is kind of a supervised learning (which is more straightforward than RL, and learning the value function). Also, the parameters of the function approximator used to model the environment can be shared with the value function. Hence these parameters can be learned simultaneously with the value function, or they can be transfered in order to increase the efficiency of computations, and enable scaling of the representation learning process to more complex tasks.

The model-free and model-based frameworks can be extended to HRL methods. In model-free HRL, as we discussed in this chapter, the agent should learn representations of the value function by integrating temporal abstraction, intrinsic motivation learning, and subgoal discovery. The model-free HRL agent maintains two (or more) value functions, one for the top-level learner (meta-controller), and one for the low-level learner (controller). Also, we need a subgoal discovery mechanism in order to discover useful subgoals and find the underlying structure of the state space.

In Model-Based HRL (MBHRL) the agent can use planning as part of the learning process. The agent can additionally learn a model that incorporates a prediction of the subgoals' distribution. While learning these models increases the computational complexity of the model-based approach, an agent equipped with a subgoal prediction mechanism might learn faster. The ability to plan hierarchically might have a positive impact on planning performance, as well as the learning of useful representations.

Learning representations in the model-based Hierarchical Reinforcement Learning framework is an interesting and open problem for a future work. Knowledge extracted from model-free HRL might be used to guide the learning of representations in model-based HRL. The collaboration of model-free HRL (direct learning) and model-based HRL (planning) is also another interesting open problem. Efficient exploration is essential for model-based RL for learning the models. We showed that intrinsic motivation learning helps with an efficient exploration in the environment with sparse delayed feedback, such as Montezuma's Revenge. We hypothesize that a behavior policy derived from intrinsic motivation learning might help with an efficient exploration in the model-based HRL framework, as well. As future work, we will consider integrating model-free HRL and model-based HRL in order to solve tasks that are too complex for a model-free RL approach alone or a model-based RL approach alone.

### 4.11.2 Solving Montezuma's Revenge

We hypothesize that integrating our model-free method with a model-based approach can help solve complex games, such as Montezuma's Revenge and Star Craft II.

Here we present an idea to solve the entire game of Montezuma's Revenge in the unified HRL framework. The game has nine different levels, each consisting of 100 screens, organized as a pyramid with a base of 19 screens wide and with 10 screens height (for more details visit `https://symlink.dk/nostalgia/c64/montezuma/`). In order to solve the game, the agent should maintain a map of the game as part of the model of the environment. In particular, the agent should know which room it is in and recognize previously visited rooms. It should also recognize when it has entered a new room as it explores the environment. It is to be noted that this map is a small graph representing the rooms' connectivity and the subgoals, and the map does not necessarily need a huge memory. The transitioning from one screen to another screen can be recognized by computing the distance between two consecutive states, $s$ $s'$, i.e. $\|s - s'\|$. When transitioning to the new screen, this distance will be large introducing an anomaly, which our unsupervised subgoal discovery process is able to discover. In some of the screens (rooms), there are rewarding objects, such as keys, doors, swords, etc. The unsupervised subgoal discovery method in concert with intrinsic motivation learning can be used to discover these rewarding objects. The agent can memorize rewarding transitions, in order to construct a model of subgoal discovery to predict the location of rewarding objects in new screens. Then the state transition model can be used to plan a path towards reaching these subgoals. The direct interaction with the environment (experience) can be used to reduce the prediction errors of the models. The agent can save pairs of `(room,object)` as the subgoals set.

The agent might learn this task with three levels of temporal abstraction. The top meta-controller receives the current screen, and chooses to stay in the room or go to other screen, and the bottom meta-controller can choose the best subgoal in that screen to pursue. Note that these two-level meta-controller can be combined, and

share parameters. The controller, then receives the room, and the subgoal location, and navigates the man to the subgoal using the skills learned in intrinsic motivation learning phase.

This game has been recently solved fully by Ecoffet et al. (2019). Their algorithm is called Go-Explore, and it is summarized in these steps: "(1) remember previously visited states, (2) first return to a promising state (without exploration), then explore from it, and (3) solve simulated environments through any available means (including by introducing determinism), and then robustify via imitation learning" (Ecoffet et al., 2019). Explore-Go, however, does not succeed in solving the game without extensive knowledge of the domain, and it requires massive memories.

The step (2) of Go-Explore method is very similar to the anomalous subgoal discovery of our unsupervised subgoal discovery method. We hypothesize that model-based HRL in concert with model-free HRL can solve this task without the need to memorize the visited states (step (1)), the imitation learning (step (3)), and the domain knowledge.

## 4.12 Conclusions

We have proposed and demonstrated a novel model-free HRL method for subgoal discovery using unsupervised learning over a small memory of the most recent experiences (trajectories) of the agent. When combined with an intrinsic motivation learning mechanism, this method learns subgoals and skills together, based on experiences in the environment. Thus, we offer an HRL approach that does not require a model of the environment, making it suitable for larger-scale applications.

Our results show that the intrinsic motivation learning produces a good policy to explore the state space efficiently, which leads to successful subgoal discovery. Our unsupervised subgoal discovery mechanism is able to find the structure of the state space, and learns the spatial hierarchies, and the meta-controller learns the temporal hierarchies to choose subgoals in the correct order.

We hypothesize that the hippocampus, in concert with the prefrontal cortex, is playing a major role in the subgoal discovery process by replaying the memory of experiences, in order to find meaningful low dimensional representation of the state.

# Chapter 5

# Trust-Region Methods for Empirical Risk Minimization

## Abstract

Deep learning algorithms often require solving a highly nonlinear and nonconvex unconstrained optimization problem. Generally, methods for solving the optimization problems in machine learning, and in deep learning specifically, are restricted to the class of first-order algorithms, like stochastic gradient descent (SGD).

The major drawback of the SGD methods is that they have the undesirable effect of not escaping saddle-points. Furthermore, these methods require exhaustive trial-and-error to fine-tune many learning parameters. Using the second-order curvature information to find the search direction can help with more robust convergence for the nonconvex optimization problem. However, computing the Hessian matrix for large-scale problems is not computationally practical.

Alternatively, quasi-Newton methods construct an approximation of the Hessian matrix to build a quadratic model of the objective function. Quasi-Newton methods, require only first-order gradient information, like SGD, but they can result in superlinear convergence, which makes them attractive alternatives. The limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) approach is one of the most popular quasi-Newton methods that constructs positive-definite Hessian approximations. Since the true Hessian matrix is not necessarily positive definite, an extra initialization condition is required when constructing the L-BFGS matrices to avoid false negative curvature information.

In this chapter, we first propose efficient optimization methods based on L-BFGS quasi-Newton methods using line search and trust-region strategies. Our method bridges the disparity between first order methods and second order methods by using gradient information to calculate low-rank updates to Hessian approximations. We provide empirical results on the classification task of the MNIST dataset and show robust convergence with preferred generalization characteristics.

Secondly, we propose various choices for initialization methods of the L-BFGS ma-

trices within a trust-region framework. We provide empirical results on the classification task of the MNIST digits dataset to compare the performance of the trust-region algorithm with different L-BFGS initialization methods.

## 5.1 Introduction

Deep learning is becoming the leading technique for solving large-scale machine learning problems, including image classification, natural language processing, and large-scale regression tasks (Goodfellow et al., 2016). Deep learning algorithms attempt to train a function approximation (*model*), usually a convolutional neural network (CNN), over a large dataset. In most of deep learning algorithms, solving an unconstrained optimization of a highly nonlinear and nonconvex objective function of the form

$$\min_{w\in\mathbb{R}^n} \mathcal{L}(w) \triangleq \frac{1}{N}\sum_{i=1}^{N} \ell_i(w) \tag{5.1}$$

is required (Hastie et al., 2009), where $w \in \mathbb{R}^n$ is the vector of trainable parameters of the CNN model, $N$ is the size of dataset, and $\ell_i(w)$ is the error of the current model's prediction for the $i$th observation of the *training* dataset.

### 5.1.1 Existing Methods

Finding an efficient optimization algorithm for the large-scale, nonconvex problem (5.1) has attracted many researchers (Goodfellow et al., 2016). There are various algorithms proposed in the machine learning and optimization literatures to solve (5.1). Among those, one can name first-order methods such as stochastic gradient descent (SGD) methods (Robbins and Monro, 1951; Bottou, 2010; Duchi et al., 2011; Recht et al., 2011), the quasi-Newton methods (Adhikari et al., 2017; Le et al., 2011a; Erway et al., 2018; Xu et al., 2017), and also Hessian-free methods (Martens, 2010; Martens and Sutskever, 2011, 2012; Bollapragada et al., 2016).

Since, in large-scale machine learning problems usually $N$ and $n$ are very large numbers, the computation of the true gradient $\nabla\mathcal{L}(w)$ is expensive and the computation of the true Hessian $\nabla^2\mathcal{L}(w)$ is not practical. Hence, most of the optimization algorithms in the machine learning and deep learning literatures are restricted to variants of first-order gradient descent methods, such as SGD methods. SGD methods use a small random sample of data ($\mathcal{S}$) to compute an approximate of the gradient of the objective function, $\nabla\mathcal{L}^{(\mathcal{S})}(w) \approx \nabla\mathcal{L}(w)$. At each iteration of the learning update, the parameters are updated as $w_{k+1} \leftarrow w_k - \eta_k \nabla\mathcal{L}^{(\mathcal{S}_k)}(w_k)$, where $\eta_k$ is referred to as the *learning rate*.

The computational cost-per-iteration of SGD algorithms is small, making them the most widely used optimization method for the vast majority of deep learning applications. However, these methods require fine-tuning of many hyperparameters,

including the learning rates. The learning rate is usually chosen to be very small; therefore, the SGD algorithms require revisiting many epochs of data during the learning process. Indeed, it is unlikely that the SGD methods perform successfully in the first attempt at a problem, though there is recent work that addresses tuning hyperparameters automatically (see e.g., (Zeiler, 2012; Kingma and Ba, 2014)).

Another major drawback of SGD methods is that they struggle with saddle-points that occur in most nonconvex optimization problems. These saddle-points have an undesirable effect on the model's generalization ability. On the other hand, using the second-order curvature information, can help produce more robust convergence. The Newton's method, a second-order method, uses the Hessian, $\nabla^2\mathcal{L}(w)$, and the gradient to find the search direction, $p_k = -\nabla^2\mathcal{L}(w_k)^{-1}\nabla\mathcal{L}(w_k)$ and then use line-search strategy to find the step length along the search direction. The main bottleneck in second-order methods is the serious computational challenges involved in the computation of the Hessian, $\nabla^2\mathcal{L}(w)$, for large-scale problems, in which it is not practical because $n$ is large. The quasi-Newton methods and Hessian-free methods both use approaches to approximate the Hessian matrix without computing and storing the true Hessian matrix, $\nabla^2\mathcal{L}(w)$. Hessian-free methods attempt to find an approximate Newton direction by solving $\nabla^2\mathcal{L}(w_k)p_k = -\nabla\mathcal{L}(w_k)$ without forming the Hessian, using conjugate-gradient methods (Martens, 2010; Martens and Sutskever, 2011, 2012; Bollapragada et al., 2016).

Quasi-Newton methods form an alternative class of first-order methods for solving the large-scale nonconvex optimization problem in deep learning. These methods, as in SGD, require only computing the first-order gradient of the objective function. By measuring and storing the difference between consecutive gradients, quasi-Newton methods construct *quasi-Newton matrices* $\{B_k\}$ which are low-rank updates to the previous Hessian approximations for estimating $\nabla^2\mathcal{L}(w_k)$ at each iteration. They build a quadratic model of the objective function by using these quasi-Newton matrices and use that model to find a sequence of search directions that can result in superlinear convergence. Since these methods do not require the second-order derivatives, they are more efficient than Newton's method for large-scale optimization problems (Nocedal and Wright, 2006).

There are various quasi-Newton methods proposed in the literature. They differ in how they define and construct the quasi-Newton matrices $\{B_k\}$, how the search directions are computed, and how the parameters of the model are updated (Nocedal and Wright, 2006; Brust et al., 2017a,b; Le et al., 2011a).

### 5.1.2 Motivation and Objectives

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (Broyden, 1970; Fletcher, 1970; Goldfarb, 1970; Shanno, 1970) is considered the most popular quasi-Newton algorithm, which produces positive semidefinite matrix $B_k$ for each iteration. The conventional BFGS minimization employs *line-search*, which first attempts to find the search directions by computing $p_k = -B_k^{-1}\nabla\mathcal{L}(w_k)$ and then decides on the step

size $\alpha_k \in (0, 1]$ based on sufficient decrease and curvature conditions (Nocedal and Wright, 2006) for each iteration $k$ and then update the parameters $w_{k+1} = w_k + \alpha_k p_k$. The line-search algorithm first tries the unit step length $\alpha_k = 1$ and if it does not satisfy the sufficient decrease and the curvature conditions, it recursively reduces $\alpha_k$ until some stopping criteria (for example $\alpha_k < 0.1$).

Solving $B_k p_k = -\nabla\mathcal{L}(w_k)$ can become computationally expensive when $B_k$ becomes a high-rank update. The *Limited-memory* BFGS (L-BFGS) method constructs a sequence of low-rank updates to the Hessian approximation and consequently solving $p_k = B_k^{-1}\nabla\mathcal{L}(w_k)$ can be done efficiently. As an alternative to gradient descent, limited memory quasi-Newton algorithms with line search have been implemented in a deep learning setting (Le et al., 2011b). These methods approximate second derivative information improving the quality of each training iteration and circumvent the need for application specific parameter tuning.

There are computational costs associated with the satisfaction of the sufficient decrease and curvature conditions, as well as finding $\alpha_k$ using line-search methods. Also if the curvature condition does not satisfy for $\alpha_k \in (0, 1]$, the L-BFGS matrix may not stay positive definite, and the update will become unstable. On the other hand, if the search direction is rejected in order to preserve the positive definiteness of L-BFGS matrices, the progress of learning might stop or become very slow.

Trust-region methods attempt to find the search direction in a region within which they trust the accuracy of the quadratic model of the objective function. These methods not only have the benefit of being independent from the fine-tuning of hyperaparameters, but they may improve upon the training performance and the convergence robustness of the line-search methods. Furthermore, trust-region L-BFGS methods can easily reject the search directions if the curvature condition is not satisfied in order to preserve the positive definiteness of the L-BFGS matrices (Rafati et al., 2018).

In this chapter, we introduce an L-BFGS quasi-Newton method based on a *trust-region* strategy. This method is called Trust-Region Minimization Algorithm for Training Responses (TRMinATR). We implement and employ this algorithm for a classification task in the deep learning framework (Rafati et al., 2018). TRMinATR solves the associated trust-region subproblem, which can be computationally intensive in large scale problems, by efficiently computing a closed form solution at each iteration. Based on the distinguishing characteristics of trust-region algorithms, unlike line-search methods, the progress of the learning will not stop or slow down due to the occasional rejection of the undesired search directions.

In order to construct the quasi-Newton matrices at each iteration, $k$, it is required to start with an initial matrix, $B_0$, that is often set to some multiple, $B_0 = \gamma_k I$, of the identity (Nocedal and Wright, 2006). Then, once $B_0$ is given, the L-BFGS matrices $B_k$ can be constructed using the L-BFGS compact representation formula (Byrd et al., 1994; Adhikari et al., 2017). The choice of the initial quasi-Newton matrix, $B_0$, is crucial because it has a direct impact on the quality of the approximation of the Hessian (Erway et al., 2018; Wah and Chuei, 2013) and the robustness of L-BFGS convergence. L-BFGS matrices are attempting the hard task of approximating the

indefinite Hessian matrix with positive definite matrices, which might result in false negative curvature information. This motivates some researchers to prefer indefinite quasi-Newton matrices such as *Limited-memory Symmetric Rank One* (L-SR1) update over the L-BFGS. However, the L-SR1 methods, unlike L-BFGS, do not guarantee a descent direction. We hypothesize that by introducing an extra condition for safe-guarding $\gamma_k$, the false negative curvature information can be avoided to some degree when approximating the Hessian matrix in an L-BFGS framework. We note that this work builds upon the results in (Erway et al., 2018) for defining $\gamma_k$.

Next, we discuss the choices for initializing L-BFGS matrices to obtain values for the parameter $\gamma_k$ that result in better training performance and generalization of learning, without introducing significant computational cost. We define extra conditions that require solving a general eigenvalue problem of form $A^*z = \lambda B^*z$, where $A^*$ and $B^*$ are obtained from the compact representation of the L-BFGS matrix (Rafati and Marcia, 2018). Consequently, solving this general eigenvalue problem does not add significant computational cost. We test our hypothesis on a supervised learning problem, namely the classification task of MNIST handwritten digits dataset, in the trust-region L-BFGS framework.

## 5.2 Background

In this section, we will present preliminary information needed for a formal description of the unconstrained optimization problem and second order optimization methods. The majority of this section is based on Numerical Optimization book by Nocedal and Wright (2006).

### 5.2.1 Unconstrained Optimization Problem

In the unconstrained optimization problem, we want to solve the minimization problem (5.1).

$$\min_w f(w), \tag{5.2}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a smooth function. A point $w^*$ is a *global* minimizer if $f(w^*) \leq f(w)$ for all $w \in \mathbb{R}^d$. Usually $f$ is a nonconvex function and most algorithms are only able to find the *local* minimizer. A point $w^*$ is a local minimizer if there is a neighborhood $\mathcal{N}$ of $w^*$ such that $f(w^*) \leq f(w)$ for all $w \in \mathcal{N}$. For convex functions, every local minimizer is also a global minimizer, but this statement is not valid for nonconvex functions. If $f$ is twice continuously differentiable we may be able to tell that $w^*$ is a local minimizer by examining the gradient $\nabla f(w^*)$ and the Hessian $\nabla^2 f(w^*)$. Let's assume that the objective function, $f$, is smooth: the first derivative (gradient) is differentiable and the second derivative (Hessian) is continuous. To study the minimizers of a smooth function, Taylor's theorem is essential.

**Theorem 5.1** (Taylor's Theorems)**.** *Suppose that* $f : \mathbb{R}^n \to \mathbb{R}$ *is continuously differentiable. Consider* $p \in \mathbb{R}^n$ *such that* $f(w+p)$ *is well defined, then we have*

$$f(w+p) = f(w) + \nabla f(w+tp)^T p, \quad \text{for some } t \in (0,1). \tag{5.3}$$

*Also if* $f$ *is twice continuously differentiable,*

$$f(w+p) = f(w) + \nabla f(w+tp)^T p + \frac{1}{2} p^T \nabla^2 f(w+tp) p, \quad \text{for some } t \in (0,1). \tag{5.4}$$

### 5.2.2 Recognizing A Local Minimum

Necessary conditions for optimality are described by the following theorems.

**Theorem 5.2** (First-Order Necessary Conditions)**.** *If* $w^*$ *is a local minimizer of* $f$ *and* $f$ *is continuously differentiable in an open neighborhood of* $w^*$*, then* $\nabla f(w^*) = 0$*.*

$w^*$ is stationary point if $\nabla f(w^*) = 0$. Any local minimizer must be a stationary point.

**Theorem 5.3** (Second-Order Necessary Conditions)**.** *If* $w^*$ *is a local minimizer and* $\nabla^2 f$ *is continuous in an open neighborhood of* $w^*$*, then* $\nabla f(w^*) = 0$ *and* $\nabla^2 f(w^*)$ *is positive semidefinite.*

The sufficient condition to guarantee that $w^*$ is a local minimizer is described in the following theorem.

**Theorem 5.4** (Second-Order Sufficient Conditions.)**.** *Suppose that* $\nabla^2 f$ *is continuous in an open neighborhood of* $w^*$ *and* $\nabla f(w^*) = 0$ *and* $\nabla^2 f(w^*)$ *is definite positive. Then* $w^*$ *is a strict local minimizer of* $f$*.*

### 5.2.3 Main Algorithms

All algorithms for unconstrained minimization require a starting point $w_0$. Beginning at $w_0$, optimization algorithms generate a sequence of iterates $\{w_k, k = 1, 2, \dots\}$ that terminate when either no more progress can be made or when it seems that a solution point has been approximated with sufficient accuracy. There are three fundamental strategies in deciding the new iterate $w_{k+1}$ given the past iteration $x_0$ to $w_k$: the *line search*, *trust region* and *conjugate gradient* methods.

## 5.3 Optimization Strategies

In this section, we briefly introduce two optimization strategies that are commonly used in quasi-Newton methods, i.e., *line search* and *trust-region* methods (Nocedal and Wright, 2006). Both methods seek to minimize the objective function $\mathcal{L}(w)$ in (5.1) by defining a sequence of iterates $\{w_k\}$ which are governed by the search

**Algorithm 12** Line Search Method pseudo-code.

**Input:** $w_0$, tolerance $\epsilon > 0$
$k \leftarrow 0$
**repeat**
  Compute $g_k = \nabla \mathcal{L}(w_k)$
  Calculate $B_k$
  Compute search direction $p_k$ by solving (5.6)
  find $\alpha_k$ that satisfies Wolfe Conditions in (5.7)
  $k \leftarrow k + 1$
**until** $\|g_k\| < \epsilon$ or $k$ reached to max number of iterations

direction $p_k$. Each respective method is defined by its approach to computing the search direction $p_k$ so as to minimize the quadratic model of the objective function defined by

$$\mathcal{Q}_k(p) \triangleq g_k^T p + \frac{1}{2} p^T B_k p, \tag{5.5}$$

where $g_k \triangleq \nabla \mathcal{L}(w_k)$ and $B_k$ is an approximation to the Hessian matrix $\nabla^2 \mathcal{L}(w_k)$. Note that $\mathcal{Q}_k(p)$ is a quadratic approximation of $\mathcal{L}(w_k + p) - \mathcal{L}(w_k)$ based on the Taylor's expansion in (5.4).

### 5.3.1 Line Search Method

Each iteration of a line search method computes a search direction $p_k$ by minimizing a quadratic model of the objective function

$$p_k = \arg\min_{p \in \mathbb{R}^n} \mathcal{Q}_k(p), \tag{5.6}$$

and then decides how far to move along that direction. The iteration is given by $w_{k+1} = w_k + \alpha_k p_k$, where $\alpha_k$ is called the step size. If $B_k$ is a positive definite matrix, the minimizer of the quadratic function can be found as $p_k = -B_k^{-1} g_k$. The ideal choice for step size $\alpha_k > 0$ is the global minimizer of the univariate function $\phi(\alpha) = \mathcal{L}(w_k + \alpha p_k)$, but in practice $\alpha_k$ is chosen to satisfy sufficient decrease and curvature conditions, e.g., the Wolfe conditions (Wolfe, 1969; Nocedal and Wright, 2006) given by

$$\mathcal{L}(w_k + \alpha_k p_k) \leq \mathcal{L}(w_k) + c_1 \alpha_k \nabla \mathcal{L}(w_k)^T p_k, \tag{5.7a}$$
$$\nabla \mathcal{L}(w_k + \alpha_k p_k)^T p_k \geq c_2 \nabla \mathcal{L}(w_k)^T p_k, \tag{5.7b}$$

with $0 < c_1 < c_2 < 1$.

The general pseudo-code for the line search method is given in Algorithm 12 (see Nocedal and Wright (2006) for details).

### 5.3.2 Trust-Region Qausi-Newton Method

Trust-region methods generate a sequence of iterates $w_{k+1} = w_k + p_k$, where each search step, $p_k$, is obtained by solving the following trust-region subproblem:

$$p_k = \underset{p \in \mathbb{R}^n}{\operatorname{argmin}} \ \mathcal{Q}_k(p) \text{ s.t. } \ \|p\|_2 \leq \delta_k, \tag{5.8}$$

where $\delta_k > 0$ is the trust-region radius. The global solution to the trust-region subproblem (5.8) can be characterized by the optimality conditions given in the following theorem due to Gay (1981) and Moré and Sorensen (1983).

**Theorem 5.5.** *Let $\delta_k$ be a positive constant. A vector $p^*$ is a global solution of the trust-region subproblem (5.8) if and only if $\|p^*\|_2 \leq \delta_k$ and there exists a unique $\sigma^* \geq 0$ such that $B + \sigma^* I$ is positive semidefinite and*

$$(B + \sigma^* I)p^* = -g \quad \textit{and} \quad \sigma^*(\delta - \|p^*\|_2) = 0. \tag{5.9}$$

*Moreover, if $B + \sigma^* I$ is positive definite, then the global minimizer is unique.*

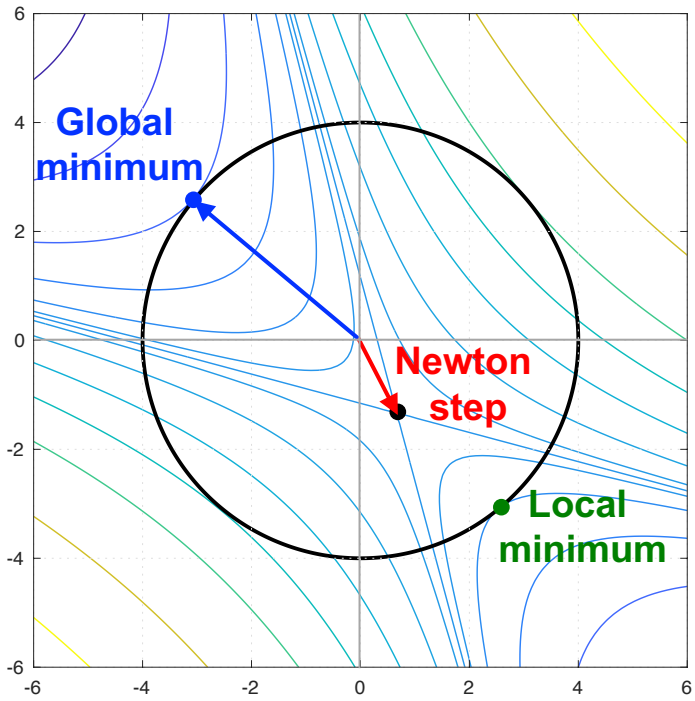



Figure 5.1: An illustration of trust-region methods. For indefinite matrices, the Newton step (in red) leads to a saddle point. The global minimizer (in blue) is characterized by the conditions in Eq. (5.9) with $B + \sigma^* I$ positive semidefinite. In contrast, local minimizers (in green) satisfy Eq. (5.9) with $B + \sigma^* I$ not positive semidefinite.

The computational bottleneck of trust-region methods is the solution of the trust-region subproblem (5.8). However, recent work (see Brust et al. (2017b) and Burdakov

**Algorithm 13** Trust region method pseudo-code.

**Input:** $w_0$, $\epsilon > 0$, $\hat{\delta} > 0$, $\delta_0 \in (0, \hat{\delta})$, $\eta \in [0, 1/4)$
$k \leftarrow 0$
**repeat**
  Compute $g_k = \nabla \mathcal{L}(w_k)$
  Construct quasi-Newton matrix $B_k$
  Compute search direction $p_k$ by solving (5.8)
  $ared \leftarrow \mathcal{L}(w_k) - \mathcal{L}(w_k + p_k)$
  $pred \leftarrow -\mathcal{Q}_k(p_k)$
  $\rho_k \leftarrow ared/pred$
  Update trust-region radius $\delta_k$
  **if** $\rho_k > \eta$ **then**
    $w_{k+1} = w_k + p_k$
  **else**
    $w_{k+1} = w_k$
  **end if**
  $k \leftarrow k + 1$
**until** $\|g_k\| < \epsilon$

et al. (2016)) has shown that (5.8) can be efficiently solved if the Hessian approximation, $B_k$, is chosen to be a quasi-Newton matrix, which we describe next. (For further details on trust-region methods, see Conn et al. (2000).)

The general pseudo-code for the trust region method is given in Algorithm 13. (see Algorithm 6.2 of Nocedal and Wright (2006) for details).

## 5.4 Quasi-Newton Optimization Methods

Methods that use $B_k = \nabla^2 \mathcal{L}(w_k)$ for the Hessian in the quadratic model in (5.5) typically exhibit quadratic rates of convergence. However, in large-scale problems (where $n$ and $N$ are both large), computing the true Hessian explicitly is not practical. In this case, quasi-Newton methods are viable alternatives because they exhibit super-linear convergence rates while maintaining memory and computational efficiency. Instead of the true Hessian, quasi-Newton methods use an approximation, $B_k$, which is updated after each step to take into account the additional knowledge gained during the step.

Quasi-Newton methods, like gradient descent methods, require only the computation of first-derivative information. They can construct a model of the objective function by measuring the changes in the consecutive gradients for estimating the Hessian. Most methods store the displacement, $\mathbf{s}_k \triangleq w_{k+1} - w_k$, and the change of gradients, $\mathbf{y}_k \triangleq \nabla \mathcal{L}(w_{k+1}) - \nabla \mathcal{L}(w_k)$, to construct the Hessian approximations, $\{B_k\}$. The quasi-Newton matrices are required to satisfy the secant equation, $B_{k+1}\mathbf{s}_k = \mathbf{y}_k$. Typically, there are additional conditions imposed on $B_{k+1}$, such as symmetry (since the exact Hessian is symmetric), and a requirement that the update to obtain $B_{k+1}$

from $B_k$ is low rank, meaning that the Hessian approximations cannot change too much from one iteration to the next. Quasi-Newton methods vary in how this update is defined. The matrices are defined recursively with the initial matrix, $B_0$, taken to be $B_0 = \lambda_{k+1} I$, where the scalar $\lambda_{k+1} > 0$.

### 5.4.1 The BFGS Update

Perhaps the most well-known among all of the quasi-Newton methods is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update (Liu and Nocedal, 1989; Nocedal and Wright, 2006), given by

$$B_{k+1} = B_k - \frac{1}{\mathbf{s}_k^T B_k \mathbf{s}_k} B_k \mathbf{s}_k \mathbf{s}_k^T B_k + \frac{1}{\mathbf{y}_k^T \mathbf{s}_k} \mathbf{y}_k \mathbf{y}_k^T. \tag{5.10}$$

The BFGS method generates positive-definite approximations whenever the initial approximation $B_0$ is positive definite and $s_k^T y_k > 0$.

### 5.4.2 The SR1 Update

The symmetric-rank-1 (SR1) is a simpler rank-1 update that maintains symmetry of the matrix and satisfies the secant equation. The SR1 update formula is given by

$$B_{k+1} = B_k + \frac{(\mathbf{y}_k - B_k \mathbf{s}_k)(\mathbf{y}_k - B_k \mathbf{s}_k)^T}{(\mathbf{y}_k - B_k \mathbf{s}_k)^T \mathbf{s}_k}. \tag{5.11}$$

To prevent SR1 from breaking down, the updates where $|\mathbf{s}_k^T (\mathbf{y}_k - B_k \mathbf{s}_k)|$ is close to zero should be skipped. Unlike the BFGS formula, SR1 is not guaranteed to generate positive definite matrices $B_k$. However, good numerical results and convergence properties have been reported with algorithms based on SR1 (Nocedal and Wright, 2006).

### 5.4.3 Compact Representations

Both BFGS and SR1 updates are low rank, and their matrices can be represented in a compact form

$$B_k = B_0 + \Psi_k M_k \Psi_k^T. \tag{5.12}$$

To compute a compact representation, we store $\mathbf{s}_k$ and $\mathbf{y}_k$ into columns of matrices $S_k$, and $Y_k$

$$S_k \triangleq [\mathbf{s}_0 \quad \dots \quad \mathbf{s}_{k-1}], \; Y_k \triangleq [\mathbf{y}_0 \quad \dots \quad \mathbf{y}_{k-1}]. \tag{5.13}$$

$\Psi_k$ and $M_k$ for BFGS are defined as

$$\Psi_k = \Big[ B_0 S_k \; Y_k \Big], \quad M_k = \begin{bmatrix} -S_k^T B_0 S_k & -L_k \\ -L_k^T & D_k \end{bmatrix}^{-1}, \tag{5.14}$$

and $L_k$ is the strictly lower triangular part and $D_k$ is the diagonal part of the matrix $S_k^T Y_k$, i.e., $S_k^T Y_k = L_k + D_k + U_k$, where $U_k$ is a strictly upper triangular matrix. (See Byrd et al. (1994) for further details.) $\Psi_k$ and $M_k$ for SR1 are defined as in Brust et al. (2017b)

$$\Psi_k = Y_k - B_0 S_k, \tag{5.15a}$$

$$M_k = (D_k + L_k + L_k^T - S_k^T B_0 S_k)^{-1}. \tag{5.15b}$$

### 5.4.4 Limited-Memory quasi-Newton methods

It is common in large-scale problems to store only the $m$ most-recently computed pairs $\{(\mathbf{s}_k, \mathbf{y}_k)\}$, where typically $m \leq 100$. This approach is often referred to as *limited-memory* BFGS (L-BFGS) when the BFGS formula is used, or limited-memory SR1 (L-SR1) when the SR1 update is used.

### 5.4.5 Trust-Region Subproblem Solution

To efficiently solve the trust-region subproblem (5.8), we exploit the compact representation of the BFGS or SR1 matrix to obtain a global solution based on optimality conditions (5.9). In particular, we compute the spectral decomposition of $B_k$ using the compact representation of $B_k$. First, we obtain the $QR$ factorization of $\Psi_k = Q_k R_k$, where $Q_k$ has orthonormal columns and $R_k$ is strictly upper triangular. Then we compute the eigendecomposition of $R_k M_k R_k^T = V_k \hat{\Lambda}_k V_k^T$, so that

$$B_k = B_0 + \Psi_k M_k \Psi_k^T = \gamma_k I + Q_k V_k \hat{\Lambda}_k V_k^T Q_k^T. \tag{5.16}$$

Note that since $V_k$ is an orthogonal matrix, the matrix $Q_k V_k$ has orthonormal columns. Let $P = [\, Q_k V_k \;\; (Q_k V_k)^\perp \,] \in \mathbb{R}^{n \times n}$, where $(Q_k V_k)^\perp$ is a matrix whose columns form an orthonormal basis for the orthogonal complement of the range space of $Q_k V_k$, thereby making $P$ an orthonormal matrix. Then

$$B_k = P \begin{bmatrix} \hat{\Lambda} + \gamma_k I & 0 \\ 0 & \gamma_k I \end{bmatrix} P^T. \tag{5.17}$$

Using this eigendecomposition to change variables and diagonalize the first optimality condition in (5.9), a closed form expression for the solution $p_k^*$ can be derived.

The general solution for the trust-region subproblem using the Sherman-Morrison-Woodbury formula is given by

$$p_k^* = -\frac{1}{\tau^*} \left[ I - \Psi_k (\tau^* M_k^{-1} + \Psi_k^T \Psi_k)^{-1} \Psi_k^T \right] g_k, \tag{5.18}$$

where $\tau^* = \gamma_k + \sigma^*$, and $\sigma^*$ is the optimal Lagrange multiplier in (5.9) (see Brust et al. (2017b) for details).

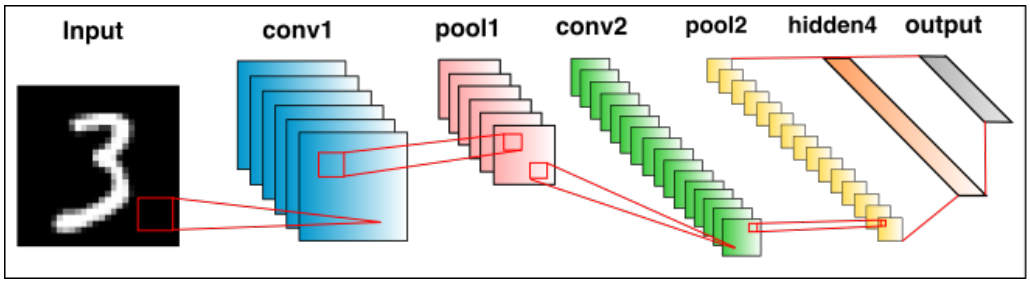



Figure 5.2: A LeNet deep learning network inspired by the architecture found in LeCun and Others (2015) . The neural network is used in the classification of the MNIST dataset of hand written digits. The convolutional neural network (CNN) uses convolutions followed by pooling layers for feature extraction. The final layer transforms the information into the required probability distribution.

## 5.5 Experiment on L-BFGS Line Search vs. Trust Region

In this section, we compare the line search L-BFGS optimization method with our proposed Trust-Region Minimization Algorithm for Training Responses (TRMinATR). The goal of the experiment is to perform the optimization necessary for neural network training. Both methods are implemented to train the LeNet-5 architecture with the purpose of image classification of the MNIST dataset. All simulations were performed on an AWS EC2 p2.xlarge instance with 1 Tesla K80 GPU, 64 GiB memory, and 4 Intel 2.7 GHz Broadwell processors. For the scalars $c_1$ and $c_2$ in the Wolfe line search condition, we used the typical values of $c_1 = 10^{-4}$ and $c_2 = 0.9$ (Nocedal and Wright, 2006). All code is implemented in the Python language using TensorFlow and it is available at `https://rafat.net/lbfgs-tr`.

### 5.5.1 LeNet-5 Convolutional Neural Network Architecture

We use the convolutional neural network architecture, LeNet-5 (Figure 5.2) for computing the likelihood $p_i(y_i|x_i; w_i)$. The LeNet-5 CNN is mainly used in the literature for character and digit recognition tasks (Lecun et al., 1998). The details of layers' connectivity in LeNet-5 CNN architecture is given in Table 5.1. The input to the network is $28 \times 28$ image and the output is 10 neurons followed by a softmax function, attempting to approximate the posterior probability distribution $p(y_i|x_i; w)$. There are a total of $n = 431,080$ trainable parameters (weights) in LeNet-5 CCN.

### 5.5.2 MNIST Image Classification Task

The convolutional neural network was trained and tested using the MNIST Dataset (LeCun, 1998). The dataset consists of 70,000 examples of handwritten digits with 60,000 examples used as a training set and 10,000 examples used as a test set. The

Table 5.1: LeNet-5 CNN architecture (Lecun et al., 1998).

| Layer | Connections |
|---|---|
| 0: input | $28 \times 28$ image |
| 1 | convolutional, 20 $5 \times 5$ filters (stride = 1), followed by ReLU |
| 2 | max pooling, $2 \times 2$ window (stride = 2) |
| 3 | convolutional, 50 $5 \times 5$ filters (stride = 1), followed by ReLU |
| 4 | max pool, $2 \times 2$ window (stride = 2) |
| 5 | fully connected, 500 neurons (no dropout) followed by ReLU |
| 6: output | fully connected, 10 neurons followed by softmax (no dropout) |

digits range from 0 - 9 and their sizes have been normalized to 28x28 pixel images. The images include labels describing their intended classification. The MNIST dataset consists of $70,000$ examples of handwritten image of digits 0 to 9, with $N = 60,000$ image training set $\{(x_i, y_i)\}$, and $10,000$ used as the test set. Each image $x_i$ is a $28 \times 28$ pixel, and each pixel value is between 0 and 255. Each image $x_i$ in the training set include a label $y_i \in \{0, \ldots, 9\}$ describing its class. The objective function for the classification task in (5.1) uses the cross entropy between model prediction and true labels given by

$$\ell_i(w) = -\sum_{j=1}^{J} y_{ij} \log(p_i), \tag{5.19}$$

where the $p_i(x_i; w) = p_i(y = y_i | x_i; w)$ is the probability distribution of the model, i.e., the likelihood that the image is correctly classified, $J$ is the number of classes ($J = 10$ for MNIST digits dataset) and $y_{ij} = 1$ if $j = y_i$ and $y_{ij} = 0$ if $j \neq y_i$ (see Hastie et al. (2009) for details).

### 5.5.3 Results

The line search algorithm and TRMinATR perform comparably in terms of loss and accuracy. This remains consistent with different choices of the memory parameter $m$ (see Fig. 5.4). The more interesting comparison is that of the training accuracy and the test accuracy. The two metrics follow each other closely. This is unlike the typical results using common gradient descent based optimization. Typically, the test accuracy is delayed in achieving the same results as the train accuracy. This would suggest that the model has a better chance of generalizing beyond the training data.

We also report that the TRMinATR significantly improves on the computational efficiency of the line-search method when using larger batch sizes. This could be

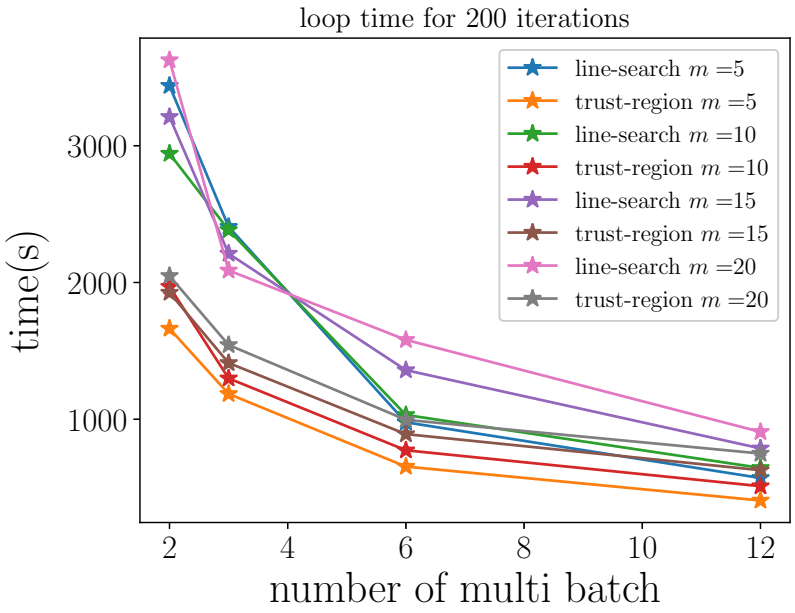



Figure 5.3: We compare the loop time for 200 iterations of the line-search and trust-region quasi-Newton algorithms for different batch sizes. As the number of multi batches increase, the size of each batch decreases. Both methods were tested using different values of the memory parameter $m$.

the result of the line-search method's need to satisfy certain Wolfe conditions at each iteration. There is also an associated computational cost when verifying that the conditions for sufficient decrease are being met. When the batch size decreases, the trust-region method continues to outperform the line-search method. This is especially true when less information is used in the Hessian approximation (see Fig. 5.3).

## 5.6 Proposed Quasi-Newton Matrix Initializations

The most common choice for initializing quasi-Newton methods is a scalar multiple of the identity matrix, i.e., $B_0 = \gamma_k I$ for $\gamma_k > 0$. In this section, we examine three choices for the scalar parameter $\gamma_k$. In particular, we label these choices as **Method I**, **Method II**, and **Method III**.

### 5.6.1 Initialization Method I

A conventional method to choose $\gamma_k$ for L-BFGS is

$$\gamma_k = \frac{y_{k-1}^T y_{k-1}}{s_{k-1}^T y_{k-1}}. \tag{5.20}$$

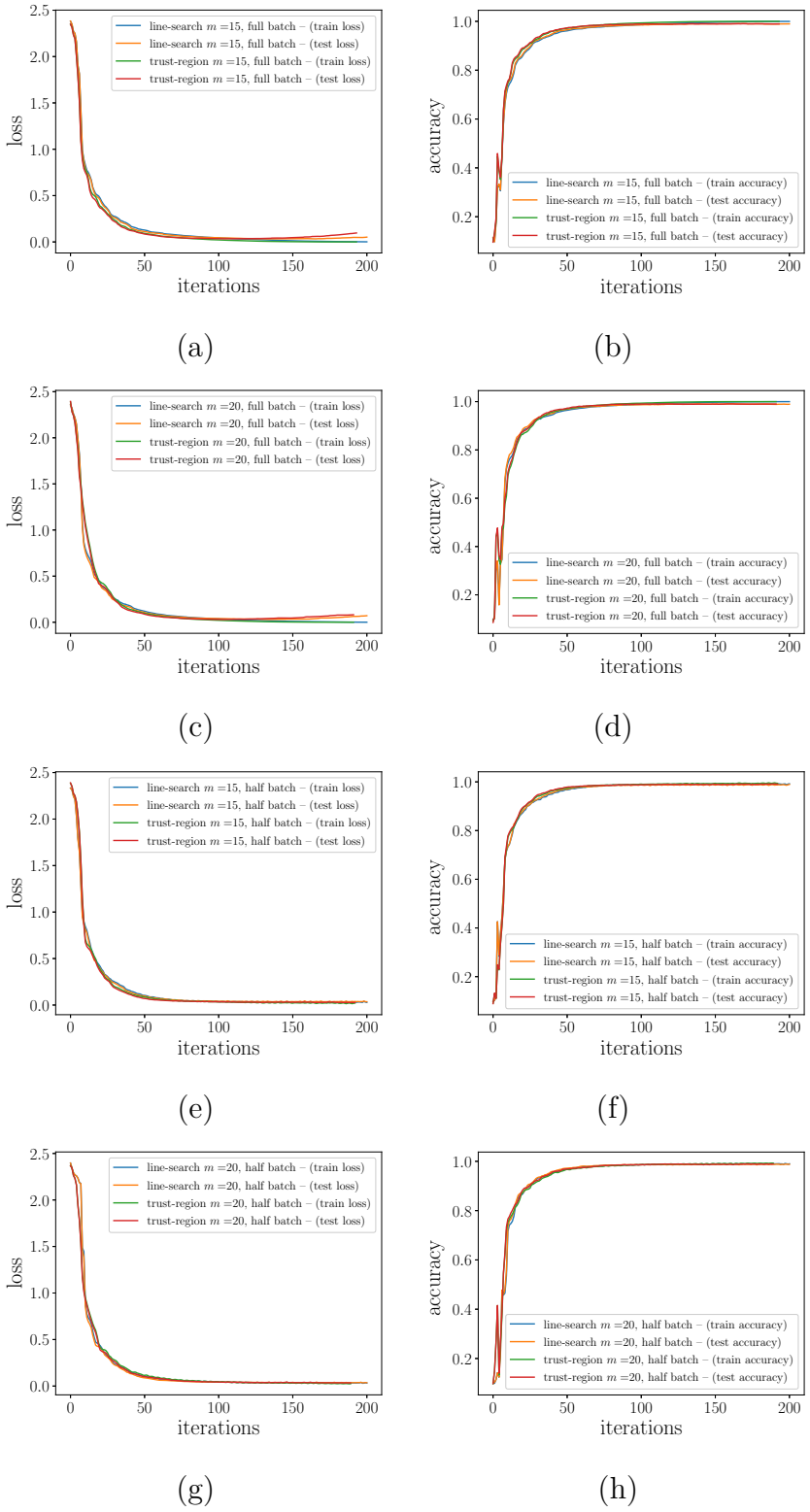



Figure 5.4: Loss and accuracy for the training and test sets, using L-BFGS line-search, and L-BFGS trust-region methods. (a) & (b) $m = 15$, and full-batch (all data is used to compute the gradients at each iteration). (c) & (d) $m = 15$, and Full-batch. (e) & (f) $m = 15$, and half-batch. (g) & (h) $m = 20$, and half-batch.

Table 5.2: Summary of the proposed L-BFGS initialization Methods

| Initialization | Formula |
|---|---|
| **Method I** | Solve the optimization problem (5.21):<br>$\gamma_k = \max\left\{1, \frac{y_{k-1}^T y_{k-1}}{s_{k-1}^T y_{k-1}}\right\}$<br>$\gamma_k = \arg\min_\gamma \|B_0^{-1} y_{k-1} - s_{k-1}\|_2^2$ |
| **Method II** | Solve the generalized eigenvalue problem (5.25):<br>$(L_k + D_k + L_k^T)z = \lambda S_k^T S_k z$<br>$\gamma_k = \begin{cases} \max\{1, 0.9\lambda_{\min}\} & \text{if } \lambda_{\min} > 0, \\ \text{Use Method I} & \text{if } \lambda_{\min} \leq 0. \end{cases}$ |
| **Method III** | Solve the generalized eigenvalue problem (5.29):<br>$A^* z = B^* \lambda z$<br>$\gamma_k = \begin{cases} \max\{1, 0.9\lambda_{\min}\} & \text{if } \lambda_{\min} > 0, \\ \text{Use Method I} & \text{if } \lambda_{\min} \leq 0. \end{cases}$ |

This choice is proposed for optimal conditioning, and it can be viewed as a spectral estimate for Hessian $\nabla^2 \mathcal{L}(w_k)$. The parameter $\gamma_k$ is the minimizer of the optimization problem

$$\gamma_k = \arg\min_\gamma \|B_0^{-1} y_{k-1} - s_{k-1}\|_2^2, \tag{5.21}$$

where $B_0^{-1} = \gamma^{-1} I$. We put a lower bound on $\gamma_k = \max(\varepsilon, \gamma_k)$, where $\varepsilon > 0$, to avoid producing sequences of nearly singular quasi-Newton matrices (Brust et al., 2017a). In our experiments, we used $\varepsilon = 1.0$.

### 5.6.2 Initialization Method II

The second method for finding $\gamma_k$ for initialization of $B_0 = \gamma_k I$ requires solving a general eigenvalue problem. This method is inspired by (Erway et al., 2018) where $\gamma_k$ is chosen in a way that avoids the false curvature information for the limited-memory Symmetric Rank-1 (L-SR1) trust-region method. We summarize the method described in Erway et al. (2018) below.

Consider a quadratic objective function of the form

$$\mathcal{L}(w) = \frac{1}{2} w^T H w + g^T w, \tag{5.22}$$

where $H \in \mathbb{R}^{n \times n}$ is symmetric and $w, g \in \mathbb{R}^n$. The true Hessian, $\nabla^2 \mathcal{L}(w)$, is equal to

matrix $H$. Note that for this quadratic function, we have

$$\nabla\mathcal{L}(w_{k+1}) - \nabla\mathcal{L}(w_k) = Hw_{k+1} + g - (Hw_k - g) = H(w_{k+1} - w_k). \tag{5.23}$$

Equivalently, $y_k = Hs_k$ for all $k$. Therefore, $HS_k = Y_k$ and consequently, $S_k^T HS_k = S_k^T Y_k$. Using the compact representation of $B_k$ in (5.12) with $B_0 = \gamma_k I$ for this quadratic function, we obtain

$$S_k^T HS_k - \gamma_k S_k^T S_k = S_k^T \Psi_k M_k \Psi_k^T S_k. \tag{5.24}$$

Note that if $H$ is not positive definite, then $S_k^T \Psi_k M_k \Psi_k^T S_k$ may not be positive definite either. Therefore, by choosing $\gamma_k > 0$, negative curvature information of $H$ can be captured by $S_k^T \Psi_k M_k \Psi_k^T S_k$. If $H$ is positive definite and $\gamma_k$ is chosen too big, then false negative curvature information can be produced. To avoid this undesired outcome, we choose $\gamma_k \in (0, \lambda_{\min})$ where $\lambda_{\min}$ is the minimum eigenvalue of the following generalized eigenvalue problem:

$$(L_k + D_k + L_k^T)z = \lambda S_k^T S_k z, \tag{5.25}$$

where $L_k$ is the strictly lower triangular part, and $D_k$ is the diagonal part of the matrix $S_k^T Y_k$ as defined in Section 5.4.3. If the smallest eigenvalue in (5.25) is negative, i.e., $\lambda_{\min} < 0$, we choose $\gamma_k$ from Method I instead.

### 5.6.3 Initialization Method III

We note that in (5.24), the right-hand side also depends on $\gamma_k$ because the matrices $\Psi_k$ amd $M_k$ depend on $\gamma_k$ (see (5.14)). Yet the generalized eigenvalue value problem (5.25) for determining bounds on $\gamma_k$ does not take this into account. In Method III, we attempt to derive the generalized eigenvalue problem considering the dependency of matrices $M_k$ and $\Psi_k$ on $\gamma_k$ as defined in (5.14).

First, we compute the inverse of $M_k$ explicitly using the following block partitioning:

$$M_k = \begin{bmatrix} -\gamma S_k^T S_k & -L_k \\ -L_k^T & D_k \end{bmatrix}^{-1} = \begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{B}^T & \tilde{D} \end{bmatrix} \tag{5.26}$$

where $\tilde{A}$, $\tilde{B}$, and $\tilde{D}$ are computed as follows:

$$\begin{aligned} \tilde{A} &= -(\gamma S_k^T S_k + L_k D_k^{-1} L_k^T)^{-1} \\ \tilde{B} &= -(\gamma S_k^T S_k + L_k D_k^{-1} L_k^T)^{-1} L_k D_k^{-1} \\ \tilde{D} &= D_k^{-1} - D_k^{-1} L_k^T (\gamma S_k^T S_k + L_k D_k^{-1} L_k^T) L_k D_k^{-1}. \end{aligned} \tag{5.27}$$

By substituting $M_k$ from (5.26) and $\Psi_k = [\gamma S_k \; Y_k]$ into (5.24), we have

$$\begin{aligned} S_k^T HS_k = \gamma S_k^T S_k + S_k^T Y_k \tilde{D} Y_k^T S_k + \gamma^2 \Big( S_k^T S_k \tilde{A} S_k^T S_k \Big) \\ + \gamma \Big( S_k^T S_k \tilde{B} Y_k^T S_k^T + S_k^T Y_k \tilde{B}^T S_k^T S_k \Big). \end{aligned} \tag{5.28}$$

The last two terms in (5.28) depend nonlinearly on $\gamma$. To find a linear condition for safe-guarding $\gamma_k$, we compute these nonlinear terms in (5.28) using the $\gamma$ parameter from the previous iteration, i.e. $\gamma_{k-1}$ (with initial value of $\gamma_0 = 1$). Then, to find an upper bound for $\gamma_k$, we solve the following generalized eigenvalue problem:

$$A^* z = \lambda B^* z, \tag{5.29}$$

where $A^*$ and $B^*$ is defined as

$$\begin{aligned} A^* &= L_k + D_k + L_k^T - S_k^T Y_k \tilde{D} Y_k^T S_k - \gamma_{k-1}^2 (S_k^T S_k \tilde{A} S_k^T S_k), \\ B^* &= S_k^T S_k + S_k^T S_k \tilde{B} Y_k^T S_k^T + S_k^T Y_k \tilde{B}^T S_k^T S_k. \end{aligned} \tag{5.30}$$

As in Method II, if $\lambda_{\min} < 0$ in (5.29), we choose $\gamma_k$ from Method I.

## 5.7 Experiments on L-BFGS Initialization

In this section, we test the trust-region L-BFGS optimization algorithm on the image classification task of the MNIST dataset with the three different initialization methods for $B_0$ discussed in Section 5.6. All simulations were performed on a cluster with one NVIDIA Tesla K20m GPU, 256 GB memory, and 20 virtual Intel 1.2 GHz processors.

All methods for this experiment are implemented to train the LeNet-5 convolutional neural network for the image classification task of the MNIST dataset. See Figure 5.2, and Table 5.1 for LeNet-5 architecture. For details on MNIST image classification task see subsection 5.5.2.

All code is implemented in the Python language using TensorFlow, NumPy and SciPy libraries and it is available at `http://rafati.net/l-bfgs-tr-init-methods`.

### 5.7.1 Computing Gradients

Computing the gradient, $\nabla \mathcal{L}(w)$, can be expensive when the size of the dataset is large. In addition, some of the data points are similar, and consequently, usually a smaller random sample $\mathcal{S}$ can be used to estimate the loss and the gradients

$$\mathcal{L}(W) \approx \mathcal{L}^{(\mathcal{S})}(w) = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \ell_i(w), \tag{5.31}$$

$$\nabla \mathcal{L}(w) \approx \nabla \mathcal{L}^{(\mathcal{S})}(w) = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \nabla \ell_i(w) \tag{5.32}$$

where, $\ell_i$ is the cross entropy between model prediction, $p_i$ and true labels, $y_i$ and $\mathcal{S}$ is a random subset of indices from $\{1, 2, \ldots, N\}$.

### 5.7.2 Multi-batch Sampling

The quality of the gradients directly impacts the quality of the search step and also the quality of approximation of the Hessian matrix. We performed our experiments using

different data-to-sample ratio $N/|\mathcal{S}| \in \{1, 2.5, 5, 12.5, 25, 50, 100, 250, 500, 1000\}$. In particular, the smaller $N/|\mathcal{S}|$ becomes, the larger the batch size, $|\mathcal{S}|$ becomes. There were an overlap between consecutive samples $\mathcal{S}_k$ and $\mathcal{S}_{k+1}$ for the each iteration of the trust-region algorithm (Algorithm 13). For iteration $k$, we used $\mathcal{S}_k$ to compute the gradient $g_k = \nabla\mathcal{L}^{(\mathcal{S}_k)}(w_k)$.

### 5.7.3 Computing $y_k$

Inspired by Berahas et al. (2016), we use the overlap between the consecutive multi-batch samples $\mathcal{O}_k = \mathcal{S}_k \cap \mathcal{S}_{k+1}$ to compute $y_k$ as

$$y_k = \nabla\mathcal{L}^{(\mathcal{O}_k)}(w_{k+1}) - \nabla\mathcal{L}^{(\mathcal{O}_k)}(w_k). \tag{5.33}$$

The use of overlap to compute $y_k$ has been shown to result in more robust convergence in L-BFGS since L-BFGS uses gradient differences to update the Hessian approximations (see Berahas et al. (2016); Erway et al. (2018)).

### 5.7.4 Other Parameters

We performed the experiments for two choices of the quasi-Newton memory storage $m \in \{10, 20\}$. All simulations stopped after 300 iterations or if the gradient satisfied $\|g_k\|_2 < \epsilon = 10^{-5}$.

### 5.7.5 Results and Discussions

The results of training using the trust-region L-BFGS algorithm with different initialization methods (**Method I**, **Method II**, and **Method III**), different multi-batch samples ($1 \leq N/|\mathcal{S}| \leq 1000$), and different memory sizes, $m$, are depicted in Fig. 5.5. For each simulation, the training and test losses, the training and test accuracy, and the total time of simulation were stored. The minimum losses for both training and test sets for $m = 10$ are plotted in Fig. 5.5(a), for different sample sizes. Note that $N/|\mathcal{S}|$ increases from left to right, meaning that the multi-batch sample size is becoming smaller. For larger sample sizes (e.g., for smaller values of $N/|\mathcal{S}|$), the initialization Method I, which is commonly used in the literature, performs the best. However, as the mini-batch sample size decreases (e.g., for $N/|\mathcal{S}| > 100$), Methods II and III outperforms Method I. For instance, the minimum training loss for Method II for $N/|\mathcal{S}| = 500$ is 2.87 times (287%) lower than the one for Method I, and the test loss for the same simulation is 120% lower. For $N/|\mathcal{S}| = 1000$, the training loss is 286% lower and the test loss is $\sim$ 79% lower. The training loss for Method III is 131% lower than the one for Method I for $N/|\mathcal{S}| = 500$, and it is 450% lower for $N/|\mathcal{S}| = 1000$. The test loss is 58% lower for $N/|\mathcal{S}| = 500$ and 100% lower for $N/|\mathcal{S}| = 1000$.

Similar phenomena can be observed for the training and test loss for $m = 20$, which is plotted in Fig. 5.5(b). The minimum training loss for simulations with initialization

Method II is $\sim$ 846% lower for $N/|\mathcal{S}| = 500$, and test loss is $\sim$ 178% lower. The training loss with Method III is $\sim$ 279% lower than Method I for $N/|\mathcal{S}| = 250$ and the test loss is $\sim$ 98% lower. The training loss with Method III is $\sim$ 426% lower for $N/|\mathcal{S}| = 500$ and the test loss is $\sim$ 125% lower.

The training and test maximum accuracy for $m = 10$ is reported in Fig. 5.5(c), and we see similar improvements using Methods II and III for simulations with smaller multi-batch sample sizes ($N/|\mathcal{S}| \geq 100$). Our proposed initialization methods results improves the test accuracy of prediction from 94.3% using Method I to 97.4% using Method II and to 96.3% using Method III, when $N/|\mathcal{S}| = 500$. We saw similar behavior in the maximum train and test accuracy for $m = 20$, which is plotted in Fig. 5.5(d). The maximum test accuracy improved from 93% using Method I to 97.5% when we used our proposed Method II, and we saw an improvement to 96.8% when we used the proposed initialization Method III.

The total training time for $m = 10$ is reported in Fig. 5.5(e). There is only $\sim$1% average increase in training time for simulations using Method II, and $\sim$ 10% average increase in training time for simulations using Method III, in comparison to Method I. Similarly, for the training run time for simulations with larger storage memory $m = 20$ in Fig. 5.5(f), there is no significant difference between Method II and Method I, but Method III was about 10% slower than the Method I.

The initialization Method I given in (5.20), which is conventionally used in the literature (Nocedal and Wright, 2006; Brust et al., 2017a), is simple to compute and usually is a great choice when the sample size is considerably large, i.e. ($|\mathcal{S}|/N \geq 1\%$ or $N/|\mathcal{S}| \leq 100$). However, once the sample sizes gets smaller, i.e. ($|\mathcal{S}|/N < 1\%$ or $N/|\mathcal{S}| > 100$), the performance drops dramatically.

Our proposed initialization Method II (based on Erway et al. (2018)) introduces a new condition on safeguarding $\gamma_k$ by finding an upper bound which requires solving a low-rank general eigenvalue problem which does not add significant computational cost. For smaller sample sizes ($|\mathcal{S}|/N < 1\%$ or $N/|\mathcal{S}| > 100$), this initialization method outperformed Method I in all training and testing minimum loss and maximum accuracy performance measures. However this method does not consider the fact that $\Psi_k$ and $M_k$ are functions of $\gamma_k$ when constructing the general eigenvalue problem in (5.25).

Our proposed initialization Method III introduces a more sophisticated condition on safeguarding $\gamma_k$. The key difference between this method and Method II is that this method takes into account that $\Psi_k$ and $M_k$ are functions of $\gamma_k$ when defining the generalized eigenvalue problem in (5.29). The computation of $A^*$ and $B^*$ adds about 10% to the computational cost. For smaller sample sizes i.e. ($|\mathcal{S}|/N < 1\%$ or $N/|\mathcal{S}| > 100$) this initialization method also outperformed the initialization Method I in all training and testing minimum loss and maximum accuracy performance measures. There was no significant difference in performance of the training when using initialization Methods II and III.

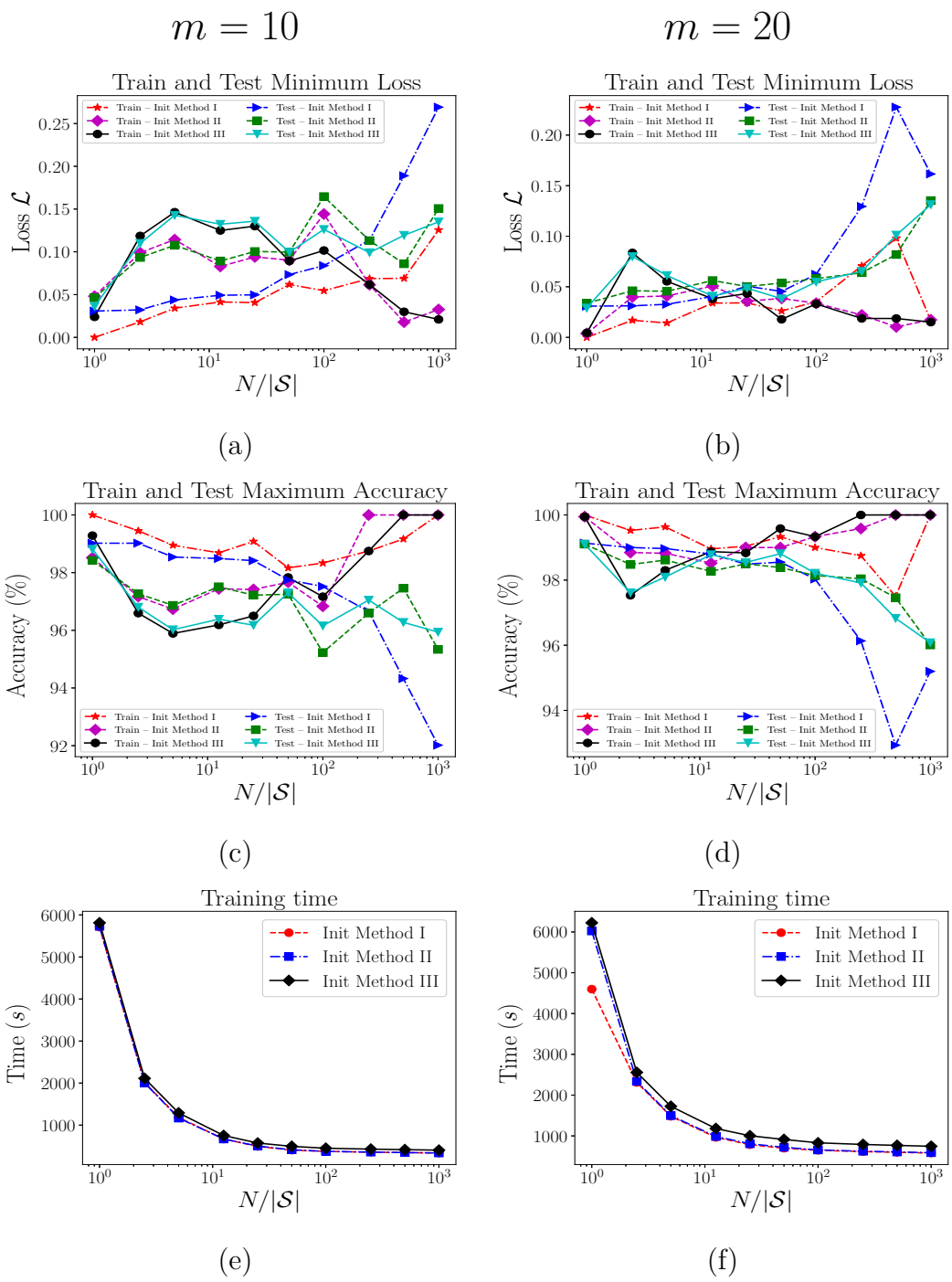



Figure 5.5: A trust-region algorithm with different L-BFGS initialization methods is used for training LeNet-5 CNN to learn the task of classification of the MNIST digits set. The performance of learning is depicted for different sample batch sizes $\mathcal{S}$ and different memory storage $m = 10$ and $m = 20$. $N$ is the size of data and $|\mathcal{S}|$ is the size of the sample batch. (a) Train and test minimum loss for $m = 10$. (b) Train and test minimum loss for $m = 20$. (c) Train and test maximum accuracy for $m = 10$. (d) Train and test maximum accuracy for $m = 20$. (e) Training time for $m = 10$. (f) Training time for $m = 20$.

## 5.8 Future Work

The true Hessian is indefinite, and using indefinite quasi-Newton matrices, like Symmetric Rank 1 (SR1), or Full Broyden Class (FBC) within trust-region methods might lead to better convergence properties. We will study these methods in a future work.

## 5.9 Conclusions

In this chapter, we implemented an optimization method based on the limited memory quasi-Newton method known as L-BFGS as an alternative to the gradient descent methods typically used to train deep neural networks. We considered both line-search and trust-region frameworks. The contribution of this research is an algorithm known as TRMinATR which minimizes the cost function of the neural network by efficiently solving a sequence of trust-region subproblems using low-rank Hessian approximations. The benefit of the method is that the algorithm is free from the constraints of data specific parameters seen in traditionally used methods. TRMinATR also improves on the computational efficiency of a similar line search implementation.

We also investigated three methods for initializing L-BFGS matrices in a trust-region optimization framework. The L-BFGS quasi-Newton matrix attempts to approximate the curvature information of the Hessian matrix, $\nabla^2\mathcal{L}(w_k)$, with a positive definite quasi-Newton matrix, $B_k$. In each iteration, $k$, an initial matrix, $B_0$, is required, and the usual choice for $B_0$ is a non-negative scalar multiple of the identity matrix, i.e., $B_0 = \gamma_k I$, with $\gamma_k > 0$. The Hessian matrix, $\nabla^2\mathcal{L}(w_k)$, can be indefinite if $\mathcal{L}(w)$ is nonconvex, and a careless initialization of the quasi-Newton matrix can have the undesired effect of definiteness mismatch between the true Hessian and the quasi-Newton Hessian approximation. We investigated three methods for the initial matrix, $B_0 = \gamma_k I$. See Table 5.2 for a summary of the initialization methods. We also experimented on the effects of initialization on the performance of training the LeNet-5 CNN on the classification task of the MNIST handwritten digits dataset.

# Chapter 6

# Quasi-Newton Optimization in Deep Reinforcement Learning


## Abstract

Reinforcement Learning (RL) algorithms allow artificial agents to improve their action selections so as to increase rewarding experiences in their environments. The learning can become intractably slow as the state space of the environment grows. This has motivated methods like Q-learning to learn representations of the state by a function approximator. Impressive results have been produced by using deep artificial neural networks. However, deep RL algorithms require solving a nonconvex and nonlinear unconstrained optimization problem. Methods for solving the optimization problems in deep RL are typically restricted to the class of first-order algorithms, such as stochastic gradient descent (SGD). The major drawback of the SGD methods is that they have the undesirable effect of not escaping saddle points and their performance can be seriously obstructed by ill-conditioning. Furthermore, SGD methods require trial and error to fine-tune many learning parameters. Using second derivative information can result in improved convergence properties, but computing the Hessian matrix for large-scale problems is not practical. Quasi-Newton methods require only first-order gradient information, like SGD, but they can construct a low rank approximation of the Hessian matrix and result in superlinear convergence. The limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) approach is one of the most popular quasi-Newton methods that constructs positive definite Hessian approximations. In this chapter, we introduce an efficient optimization method, based on the limited memory BFGS quasi-Newton method using the line search strategy – as an alternative to SGD methods. Our method bridges the disparity between first order methods and second order methods by continuing to use gradient information to calculate a low-rank Hessian approximations. We provide a formal convergence analysis, as well as empirical results on a subset of the classic ATARI 2600 games. Our results show a robust convergence with preferred generalization characteristics, as well as fast training time and no need for an experience replaying mechanism.

## 6.1 Introduction

One of the challenges that arise in real-world reinforcement learning (RL) problems is the "curse of dimensionality". Nonlinear function approximators coupled with reinforcement learning have made it possible to learn abstractions over high dimensional state spaces (Sutton, 1996; Rafati and Noelle, 2015, 2017; Melo et al., 2008). Successful examples of using neural networks for reinforcement learning include learning how to play the game of Backgammon at the Grand Master level (Tesauro, 1995). More recently, researchers at DeepMind Technologies used a deep Q-learning algorithm to play various ATARI games from the raw screen image stream (Mnih et al., 2013, 2015). The Deep Q-learning algorithm (Mnih et al., 2013) employed a convolutional neural network (CNN) as the state-action value function approximation. The resulting performance on these games was frequently at or better than the human level. In another effort, DeepMind used deep CNNs and a Monte Carlo Tree Search algorithm that combines supervised learning and reinforcement learning to learn how to play the game of Go at a super-human level (Silver et al., 2016).

The majority of deep learning problems, including deep RL algorithms, require solving an unconstrained optimization of a highly nonlinear and nonconvex objective function of the form

$$\min_{w\in\mathbb{R}^n} \mathcal{L}(w) = \frac{1}{N}\sum_{i}^{N} \ell_i(w) \tag{6.1}$$

where $w \in \mathbb{R}^n$ is the vector of trainable parameters of the CNN model, $n$ is the number of parameters, $N$ is the number of observations in a training dataset (or RL agent's experience memory), and $\ell_i$ is a function of the $i$th observation. There are various algorithms proposed in machine learning and optimization literature to solve (6.1). Among those, there are first-order methods such as stochastic gradient descent (SGD) methods (Robbins and Monro, 1951) and quasi-Newton methods (Le et al., 2011b). For instance, a variant of the SGD method was used in DeepMind's implementation of deep Q-Learning algorithm (Mnih et al., 2015).

Since both $n$ and $N$ are large in large-scale problems, the computation of the true gradient, $\nabla\mathcal{L}(w)$, is expensive and, additionally, the computation of the true Hessian, $\nabla^2\mathcal{L}(w)$, is not practical. At each iteration of learning, SGD methods use a small random sample of data, $J_k$, to compute an approximation of the gradient of the objective function, $\nabla^{(J_k)}\mathcal{L}(w_k)$, and they use the opposite of that vector as the search direction, $p_k = -\nabla^{(J_k)}\mathcal{L}(w_k)$. The computational cost-per-iteration of SGD algorithms is small, making them the most widely used optimization method for the vast majority of deep learning and deep RL applications.

However, these methods require the fine-tuning of many hyperparameters, including the learning rates, $\alpha_k$. The learning rates are usually chosen to be very small to decrease the undesirable effect of the noisy stochastic gradient. Therefore, deep RL methods based on SGD algorithms require storing a large memory of recent experiences into a *experience replay memory* $\mathcal{D}$ and replaying this memory repeatedly.

Another major drawback of the SGD methods is that they struggle with saddle-points and the problem ill-conditioning that occur in most of the nonconvex optimization and has the undesirable effect on the model's generalization of learning (Bottou et al., 2018).

Using second-order curvature information can help produce more robust convergence for nonconvex optimization problems (Nocedal and Wright, 2006; Bottou et al., 2018). An example of a second-order method is Newton's method, where the Hessian matrix, $\nabla^2\mathcal{L}(w)$, and the gradient, $\nabla\mathcal{L}(w)$, are used to find the search direction, $p_k = -\nabla^2\mathcal{L}(w_k)^{-1}\nabla\mathcal{L}(w_k)$, and then a line-search method is used to find the step length along the search direction. The main bottleneck in second-order methods is the serious computational challenges involved in computing the Hessian, $\nabla^2\mathcal{L}(w)$, for deep reinforcement learning problems, which is not practical when $n$ is large. Quasi-Newton methods and Hessian-free methods both use approaches to approximate the Hessian matrix without computing and storing the true Hessian matrix, $\nabla^2\mathcal{L}(w)$.

Quasi-Newton methods form an alternative class of first-order methods for solving the large-scale nonconvex optimization problem in deep learning (Nocedal and Wright, 2006; Erway et al., 2018; Rafati et al., 2018). These methods, like SGD, require only computing the first-order gradient of the objective function. By measuring and storing the difference between consecutive gradients, quasi-Newton methods construct *quasi-Newton matrices*, $\{B_k\}$, which are low-rank updates to previous Hessian approximations for estimating $\nabla^2\mathcal{L}(w_k)$ at each iteration. They build a quadratic model of the objective function by using these quasi-Newton matrices and use that model to find a sequence of search directions that can result in superlinear convergence. Since these methods do not require the second-order derivatives, they are more efficient than Newton's method for large-scale optimization problems (Nocedal and Wright, 2006).

There are various quasi-Newton methods proposed in literature. They differ in how they define and construct the quasi-Newton matrices $\{B_k\}$, how the search directions are computed, and how the parameters of the model are updated. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (Broyden, 1970; Fletcher, 1970; Goldfarb, 1970; Shanno, 1970) is one of the most well-regarded quasi-Newton algorithm, producing a positive semidefinite matrix, $B_k$, for each iteration.

The *Limited-memory* BFGS (L-BFGS) method constructs a sequence of low-rank updates to the Hessian approximation and, consequently, solving $p_k = B_k^{-1}\nabla\mathcal{L}(w_k)$ can be done efficiently. Methods based on L-BFGS quasi-Newton have been implemented and employed for the image classification task in the deep learning framework and impressive results have been produced (Rafati et al., 2018; Rafati and Marcia, 2018; Berahas et al., 2016; Le et al., 2011a).

These methods approximate second derivative information, improving the quality of each training iteration and circumventing the need for application-specific parameter tuning. Given that quasi-Newton methods are efficient in supervised learning problems (Bottou et al., 2018), an important question arises: Is it also possible to use quasi-Newton methods to learn the state representations in deep reinforcement

learning successfully? We will investigate this question in the remainder of this chapter.

In this chapter, we implement a limited-memory BFGS (L-BFGS) optimization method for deep reinforcement learning framework. Our deep L-BFGS Q-learning method is designed to be efficient for parallel computation using GPUs. We investigate our algorithm using a subset of the ATARI 2600 games, assessing its ability to learn robust representations of the state-action value function, as well as its computation and memory efficiency. We also analyze the convergence properties of Q-learning using a deep neural network employing L-BFGS optimization.

## 6.2 Optimization Problems in RL

In an RL problem, the agent should implement a policy, $\pi$, from states, $\mathcal{S}$, to possible actions, $\mathcal{A}$, to maximize its expected return from the environment (Sutton and Barto, 2017). At each cycle of interaction, the agent receives a state, $s$, from the environment, takes an action, $a$, and one time step later, the environment sends a reward, $r \in \mathbb{R}$, and an updated state, $s'$. Each cycle of interaction, $e = (s, a, r, s')$ is called a transition *experience* (or a trajectory). The goal is to find an optimal policy that maximizes the expected value of the return, i.e. the cumulative sum of future rewards, $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'+1}$, where $\gamma \in [0, 1]$ is a discount factor, and $T$ as a final step. It is often useful to define a parametrized value function $Q(s, a; w)$ to estimate the expected value of the return. Q-learning is a Temporal Difference (model-free RL) algorithm that attempts to find the optimal value function by minimizing the loss function, $L(w)$, which is defined over a recent *experience memory* $\mathcal{D}$:

$$\min_{w \in \mathbb{R}^n} \mathcal{L}(w) \triangleq \frac{1}{2} \mathbb{E}_{e \sim \mathcal{D}} \Big[ \big( \mathcal{Y} - Q(s, a; w) \big)^2 \Big], \tag{6.2}$$

where $\mathcal{Y} = r + \max_{a'} Q(s', a'; w)$ is the target value for the expected return based on the *Bellman's optimality* equations (Sutton and Barto, 1998).

In practice, instead of minimization of the expected risk in (6.2) we can define an optimization problem for the *empirical risk* as follows

$$\min_{w \in \mathbb{R}^n} \mathcal{L}(w) \triangleq \frac{1}{2|\mathcal{D}|} \sum_{e \in \mathcal{D}} \Big[ \big( \mathcal{Y} - Q(s, a; w) \big)^2 \Big]. \tag{6.3}$$

The most common approach for solving the empirical risk minimization problem (6.3) in literature involves using a variant of the stochastic gradient decent (SGD) method (6.3). At each optimization step, $k$, a small set of experiences, $J_k$, are randomly sampled from the *experience replay memory*, $\mathcal{D}$. This sample is used to compute an stochastic gradient of the objective function, $\nabla \mathcal{L}(w)^{J_k}$, as an approximation for the true gradient, $\nabla \mathcal{L}(w)$,

$$\nabla \mathcal{L}(w)^{(J_k)} \triangleq \frac{-1}{|J_k|} \sum_{e \in J_k} \Big[ \big( \mathcal{Y} - Q(s, a; w) \big) \nabla Q \Big]. \tag{6.4}$$

The stochastic gradient then can be used to update the iterate $w_k$ to $w_{k+1}$

$$w_{k+1} = w_k - \alpha_k \nabla \mathcal{L}(w_k)^{(J_k)}, \tag{6.5}$$

where $\alpha_k$ is the learning rate (step size).

## 6.3 Line-search L-BFGS Optimization

In this section, we briefly introduce a quasi-Newton optimization method based on the *line-search* strategy, as an alternative for SGD methods. Then we introduce the limited-memory BFGS method.

### 6.3.1 Line Search Method

Each iteration of a line search method computes a search direction, $p_k$, and then decides how far to move along that direction. The iteration is given by

$$w_{k+1} = w_k + \alpha_k p_k, \tag{6.6}$$

where $\alpha_k$ is called the step size. The search direction $p_k$ is obtained by minimizing a quadratic model of the objective function defined by

$$p_k = \min_{p \in \mathbb{R}^n} q_k(p) \triangleq g_k^T p + \frac{1}{2} p^T B_k p, \tag{6.7}$$

where $g_k = \nabla \mathcal{L}(w_k) \in \mathbb{R}^n$ is the gradient of the objective function at $w_k$, and $B_k$ is an approximation to the Hessian matrix $\nabla^2 \mathcal{L}(w_k) \in \mathbb{R}^{n \times n}$. If $B_k$ is a positive definite matrix, the minimizer of the quadratic function can be calculated directly as

$$p_k = -B_k^{-1} g_k. \tag{6.8}$$

The step size, $\alpha_k$, is chosen to satisfy the Wolfe conditions (Nocedal and Wright, 2006) given by

$$\mathcal{L}(w_k + \alpha_k p_k) \leq \mathcal{L}(w_k) + c_1 \alpha_k \nabla \mathcal{L}_k^T p_k, \tag{6.9a}$$

$$\nabla \mathcal{L}(w_k + \alpha_k p_k)^T p_k \geq c_2 \nabla \mathcal{L}(w_k)^T p_k, \tag{6.9b}$$

with $0 < c_1 < c_2 < 1$. The Wolfe conditions consist of a sufficient decrease condition in (6.9a) and a curvature condition in (6.9).

### 6.3.2 Quasi-Newton Optimization Methods

Methods that use the Hessian for $B_k$, $B_k = \nabla^2 \mathcal{L}(w_k)$, in the quadratic model in (6.7) typically exhibit quadratic rates of convergence. However, in large-scale problems (where $n$ and $N$ are both large), computing the true Hessian explicitly is not

practical. In this case, quasi-Newton methods are viable alternatives because they exhibit super-linear convergence rates while maintaining memory and computational efficiency. Instead of the true Hessian, quasi-Newton methods use an approximation $B_k$, which is updated after each step to take account of the additional knowledge gained during the step.

Quasi-Newton methods, like gradient descent methods, require only the computation of first-derivative information. They can perform efficient optimization by constructing a quadratic model of the objective function, using the changes in consecutive gradients as an estimate of the Hessian. The *quasi-Newton matrices*, $\{B_k\}$, are required to satisfy the secant equation

$$B_{k+1}(w_{k+1} - w_k) \approx \nabla\mathcal{L}(w_{k+1}) - \nabla\mathcal{L}(w_k). \tag{6.10}$$

Typically, there are additional conditions imposed on $B_{k+1}$, such as symmetry (since the exact Hessian is symmetric), and a requirement that the update to obtain $B_{k+1}$ from $B_k$ is low rank, meaning that the Hessian approximations cannot change too much from one iteration to the next. Quasi-Newton methods vary in how this update is defined.

### 6.3.3 The BFGS Quasi-Newton Update

Perhaps the most well-known among all of the quasi-Newton methods is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update (Liu and Nocedal, 1989; Nocedal and Wright, 2006), given by

$$B_{k+1} = B_k - \frac{1}{\mathbf{s}_k^T B_k \mathbf{s}_k} B_k \mathbf{s}_k \mathbf{s}_k^T B_k + \frac{1}{\mathbf{y}_k^T \mathbf{s}_k} \mathbf{y}_k \mathbf{y}_k^T, \tag{6.11}$$

where $\mathbf{s}_k = w_{k+1} - w_k$ and $\mathbf{y}_k = \nabla\mathcal{L}(w_{k+1}) - \nabla\mathcal{L}(w_k)$. The matrices are defined recursively with the initial matrix, $B_0 = \lambda_{k+1} I$, where the scalar $\lambda_{k+1} > 0$. The BFGS method generates positive-definite approximations whenever the initial approximation $B_0$ is positive definite and $s_k^T y_k > 0$.

### 6.3.4 Limited-Memory BFGS

In practice, only the $m$ most-recently computed pairs $\{(\mathbf{s}_k, \mathbf{y}_k)\}$ are stored, where $m \ll n$, typically $m \leq 100$ for very large problems. This approach is often referred to as *limited-memory* BFGS (L-BFGS). Since we have to compute $p_k = -B_k^{-1} g_k$ at each iteration, we make use of the following recursive formula for $H_k = B_k^{-1}$:

$$H_{k+1} = \left(I - \frac{\mathbf{y}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\right) H_k \left(I - \frac{\mathbf{s}_k \mathbf{y}_k^T}{\mathbf{y}_k \mathbf{s}_k^T}\right) + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k \mathbf{s}_k^T}, \tag{6.12}$$

where $H_0 = \gamma_{k+1} I$. A common value for $\gamma_{k+1} > 0$ is $\mathbf{y}_k^T \mathbf{s}_k / \mathbf{y}_k^T \mathbf{y}_k$ (Nocedal and Wright, 2006; Rafati and Marcia, 2018). The *L-BFGS two-loop recursion algorithm*, given in Algorithm 14, can compute $p_k = -H_k g_k$ in $4mn$ operations (Nocedal and Wright, 2006).

**Algorithm 14** L-BFGS two-loop recursion.

$\mathbf{q} \leftarrow g_k = \nabla\mathcal{L}(w_k)$
**for** $i = k-1, \ldots, k-m$ **do**
  $\alpha_i = \frac{\mathbf{s}_i^T q}{\mathbf{y}_i^T \mathbf{s}_i}$
  $\mathbf{q} \leftarrow \mathbf{q} - \alpha_i \mathbf{y}_i$
**end for**
$\mathbf{r} \leftarrow H_0 q$
**for** $i = k-1, \ldots, k-m$ **do**
  $\beta = \frac{\mathbf{y}_i^T r}{\mathbf{y}_i^T \mathbf{s}_i}$
  $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{s}_i(\alpha_i - \beta)$
**end for**
**return** $-\mathbf{r} = -H_k g_k$

## 6.4 Deep L-BFGS Q Learning Method

In this section, we propose a novel algorithm for the optimization problem in deep Q-Learning framework, based on the limited-memory BFGS method using a line search strategy. This algorithm is designed to be efficient for parallel computations on GPU. Also the experience memory $\mathcal{D}$ is emptied after each gradient computation, hence the algorithm needs much less RAM memory.

Inspired by Berahas et al. (2016), we use the overlap between the consecutive multi-batch samples $O_k = J_k \cap J_{k+1}$ to compute $y_k$ as

$$\mathbf{y}_k = \nabla\mathcal{L}(w_{k+1})^{(O_k)} - \nabla\mathcal{L}(w_k)^{(O_k)}. \tag{6.13}$$

The use of overlap to compute $y_k$ has been shown to result in more robust convergence in L-BFGS since L-BFGS uses gradient differences to update the Hessian approximations (see Berahas et al. (2016) and Erway et al. (2018)).

At each iteration of optimization we collect experiences in $\mathcal{D}$ up to batch size $b$ and use the entire experience memory $\mathcal{D}$ as the overlap of consecutive samples $O_k$. For computing the gradient $g_k = \nabla\mathcal{L}(w_k)$, we use the $k$th sample, $J_k = O_{k-1} \cup O_k$

$$\nabla\mathcal{L}(w_k)^{(J_k)} = \frac{1}{2}(\nabla\mathcal{L}(w_k)^{(O_{k-1})} + \nabla\mathcal{L}(w_k)^{(O_k)}). \tag{6.14}$$

Since $\nabla\mathcal{L}(w_k)^{(O_{k-1})}$ is already computed to obtain $\mathbf{y}_{k-1}$ in the previous iteration, we only need to compute $\nabla\mathcal{L}^{(O_k)}(w_k)$, given by

$$\nabla\mathcal{L}(w_k)^{(O_k)} = \frac{-1}{|\mathcal{D}|}\sum_{e\in D}\Big[\Big(\mathcal{Y} - Q(s,a;w_k)\Big)\nabla Q\Big]. \tag{6.15}$$

Note that in order to obtain $\mathbf{y}_k$, we only need to compute $\nabla\mathcal{L}(w_{k+1})^{(O_k)}$ since $\nabla\mathcal{L}(w_k)^{(O_k)}$ is already computed when we computed the gradient in (6.14).

The line search multi-batch L-BFGS optimization algorithm for deep Q-Leaning is provided in Algorithm 15.

**Algorithm 15** Line search Multi-batch L-BFGS Optimization for Deep Q Learning.

**Inputs:** batch size $b$, L-BFGS memory $m$, exploration rate $\epsilon$
**Initialize** experience memory $\mathcal{D} \leftarrow \emptyset$ with capacity $b$
**Initialize** $w_0$, i.e. parameters of $Q(.,.;w)$ randomly
**Initialize** optimization iteration $k \leftarrow 0$
**for** episode $= 1, \ldots, M$ **do**
  Initialize state $s \in \mathcal{S}$
  **repeat** for each step $t = 1, \ldots, T$
    compute $Q(s, a; w_k)$
    $a \leftarrow$`EPS-GREEDY`$(Q(s, a; w_k), \epsilon)$
    Take action $a$
    Observe next state $s'$ and external reward $r$
    Store transition experience $e = \{s, a, r, s'\}$ to $\mathcal{D}$
    $s \leftarrow s'$
  **until** $s$ is terminal or intrinsic task is done
  **if** $|\mathcal{D}| == b$ **then**
    $O_k \leftarrow \mathcal{D}$
    Update $w_k$ by performing **optimization step**
    $\mathcal{D} \leftarrow \emptyset$
  **end if**
**end for**

==========

**Multi-batch line search L-BFGS Optimization step:**

Compute gradient $g_k^{(O_k)}$
Compute gradient $g_k^{(J_k)} \leftarrow \frac{1}{2} g_k^{(O_k)} + \frac{1}{2} g_k^{(O_{k-1})}$
Compute $p_k = -B_k^{-1} g_k^{(J_k)}$ using Algorithm 14
Compute $\alpha_k$ by satisfying the Wolfe Conditions (6.9)
Update iterate $w_{k+1} = w_k + \alpha_k p_k$
$\mathbf{s}_k \leftarrow w_{k+1} - w_k$
Compute $g_{k+1}^{(O_k)} = \nabla \mathcal{L}(w_{k+1})^{(O_k)}$
$\mathbf{y}_k \leftarrow g_{k+1}^{(O_k)} - g_k^{(O_k)}$
Store $\mathbf{s}_k$ to $S_k$ and $\mathbf{y}_k$ to $Y_k$ and remove oldest pairs
$k \leftarrow k + 1$

## 6.5 Convergence Analysis

In this section, we present a convergence analysis for our deep Q-learning with multi-batch line-search L-BFGS optimization method (Algorithm 15). We also provide an analysis for optimality of the state action value function. We then provide a comparison between the computation time of our deep L-BFGS Q-learning method (Algorithm 15) and that of DeepMind's Deep Q-learning algorithm (Mnih et al., 2015), which uses a variant of the SGD method.

### 6.5.1 Convergence of empirical risk

To analyze the convergence properties of empirical risk function $\mathcal{L}(w)$ in (6.3) we assume that

$$\mathcal{L}(w) \text{ is strongly convex, and twice differentiable.} \tag{6.16a}$$
$$\forall w,\ \exists \lambda, \Lambda > 0 \text{ such that } \lambda I \preceq \nabla^2 \mathcal{L}(w) \mathcal{L} \preceq \Lambda I, \text{ i.e. Hessian is bounded.} \tag{6.16b}$$
$$\forall w,\ \exists \eta > 0 \text{ such that } \|\nabla \mathcal{L}(w)\|^2 \leq \eta^2, \text{ i.e. Gradient does not explode.} \tag{6.16c}$$

**Lemma 6.1.** *$\exists \lambda', \Lambda' > 0$ such that $\lambda' I \preceq H_k \preceq \Lambda' I$.*

*Proof.* Due to the assumptions (6.16a) and (6.16b), the eigenvalues of the positive-definite matrix $H_k$ are also bounded (Byrd et al., 2016; Berahas et al., 2016). □

**Lemma 6.2.** *Let $w^*$ be a minimizer of $\mathcal{L}$. Then, for all $w$, we have $2\lambda(\mathcal{L}(w) - \mathcal{L}(w^*) \leq \|\nabla \mathcal{L}(w)\|^2$.*

*Proof.* For any convex function, $\mathcal{L}$, and for any two points, $w$ and $w^*$, one can show that (Nesterov, 2013):

$$\begin{aligned} \mathcal{L}(w) \leq \mathcal{L}(w^*) + \nabla \mathcal{L}(w^*)^T (w - w^*) \\ + \frac{1}{2\lambda} \|\nabla \mathcal{L}(w) - \nabla \mathcal{L}(w^*)\|^2. \end{aligned} \tag{6.17}$$

Since $w^*$ is a minimizer of $\mathcal{L}$ then $\nabla \mathcal{L}(w^*) = 0$ in (6.17) and we have the proof. □

**Theorem 6.1.** *Let $w_k$ be iterates generated by Algorithm 15, and let's assume that the step length, $\alpha_k$, is fixed. The upper bound for the empirical risk offset from the true minimum value is*

$$\begin{aligned} \mathcal{L}(w_k) - \mathcal{L}(w^*) \leq (1 - 2\alpha\lambda\lambda')^k [\mathcal{L}(w_0) - \mathcal{L}(w^*)] \\ + [1 - (1 - 2\alpha\lambda\lambda')^k] \frac{\alpha^2 \Lambda'^2 \Lambda \eta^2}{4\lambda'\lambda}. \end{aligned} \tag{6.18}$$

*Proof.* By using Taylor expansion on

$$\mathcal{L}(w_{k+1}) = \mathcal{L}(w_k - \alpha_k H \nabla \mathcal{L}(w_k))$$

around $w_k$ we can have

$$\begin{aligned}\mathcal{L}(w_{k+1}) \leq \mathcal{L}(w_k) - \alpha_k \nabla \mathcal{L}(w_k)^T H_k \nabla \mathcal{L}(w_k) \\ + \frac{\Lambda}{2} \|\alpha_k \nabla \mathcal{L}(w_k)^T H_k \nabla \mathcal{L}(w_k)\|^2.\end{aligned} \tag{6.19}$$

By applying assumptions (6.16) and Lemma 6.1 and 6.2 to the above inequality, we have

$$\begin{aligned}&\mathcal{L}(w_{k+1}) \leq \mathcal{L}(w_k) \\ &- 2\alpha_k \lambda' \lambda [\mathcal{L}(w_k) - \mathcal{L}(w^*)] + \frac{\alpha_k^2 \Lambda'^2 \Lambda \eta^2}{4\lambda' \lambda}\end{aligned} \tag{6.20}$$

By rearranging terms and using recursion expression and recursion over $k$ we have the proof. For a more detailed proof see Byrd et al. (2016) and Berahas et al. (2016). □

If the step size is bounded, $\alpha \in (0, 1/2\lambda\lambda')$, we can conclude that the first term of the bound given in (6.18) is decaying linearly to zero when $k \to \infty$ and the constant residual term, $\frac{\alpha^2 \Lambda'^2 \Lambda \eta^2}{4\lambda' \lambda}$, is the neighborhood of convergence.

### 6.5.2 Value Optimality

The Q-learning method has been proved to converge to the optimal value function if the step sizes satisfies $\sum_k \alpha_k = \infty$ and $\sum_k \alpha_k^2 < \infty$ (Jaakkola et al., 1994). Now, we want to prove that Q-learning using the L-BFGS update also theoretically converges to the optimal value function under one additional condition on the step length, $\alpha_k$.

**Theorem 6.2.** *Let $Q^*$ be the optimal state-action value function and $Q_k$ be the Q-function with parameters $w_k$. Furthermore, assume that the gradient of $Q$ is bounded, $\|\nabla Q\|^2 \leq \eta''^2$, and the Hessian of $Q$ functions satisfy $\lambda'' \preceq \nabla^2 Q \preceq \Lambda''$. We have*

$$\begin{aligned}&\|Q_{k+1} - Q^*\|_\infty < \\ &\prod_{j=0}^{k} \Big[1 - \alpha_j \eta''^2 \lambda + \frac{\alpha_j \eta''^2 \Lambda'^2 \Lambda''}{2}\Big]^k \|Q_0 - Q^*\|_\infty.\end{aligned} \tag{6.21}$$

*If step size $\alpha_k$ satisfies*

$$\Big|1 - \alpha_k \eta''^2 \lambda + \frac{\alpha_k \eta \eta' \Lambda'^2 \Lambda''}{2}\Big| < 1, \ \forall k, \tag{6.22}$$

*$Q(., .; w_k)$ ultimately will converge to $Q^*$, when $k \to \infty$.*

*Proof.* First we derive the effect of the parameter update from $w_k$ to

$$w_{k+1} = w_k - \alpha_k H_k \nabla \mathcal{L}(w_k)$$

on the optimality neighbor.

$$\|Q_{k+1} - Q^*\|_\infty \triangleq \max_{s,a} \Big| Q(s, a, w_{k+1}) - Q^*(s, a) \Big| \tag{6.23}$$

We approximate the gradient using only one experience $(s, a, r, s')$,

$$\nabla \mathcal{L}(w_k) \approx \Big( Q(s, a; w_k) - Q^*(s, a; w_k) \Big) \nabla Q_k(s, a; w_k), \tag{6.24}$$

Using Taylor's expansion to approximate $Q(s, a, w_{k+1})$ results in

$$\begin{aligned} &Q(s, a; w_{k+1}) = Q(s, a; w_k - \alpha_k H_k \nabla \mathcal{L}(w_k)) \\ &= Q(s, a; w_k) - \alpha_k \nabla \mathcal{L}_k^T H_k \nabla Q_k + \frac{\alpha_k^2}{2} \nabla \mathcal{L}_k^T H_k \nabla^2 Q(\xi_k) H_k \nabla \mathcal{L}_k^T \\ &= Q_k - \alpha_k (Q_k - Q^*) \nabla Q_k^T H_k \nabla Q_k + \frac{\alpha_k^2}{2} (Q_k - Q^*) \nabla Q_k^T H_k \nabla^2 Q(\xi_k) H_k \nabla \mathcal{L}_k^T, \end{aligned} \tag{6.25}$$

where $w_k < \xi_k < w_{k+1}$, $Q_k \coloneqq Q(s, a; w_k)$, $\nabla Q_k \coloneqq \nabla Q(s, a; w_k)$, and $\nabla \mathcal{L}_k \coloneqq \nabla \mathcal{L}(w_k)$. We can use the above expression to compute $\|Q_{k+1} - Q^*\|_\infty$

$$\begin{aligned} &\|Q_{k+1} - Q^*\|_\infty = \\ &\max_{s,a} \Big| (Q_k - Q^*) \Big[ 1 - \alpha_k \nabla Q_k^T H_k \nabla Q_k + \frac{\alpha_k^2}{2} \nabla Q_k^T H_k \nabla^2 Q(\xi_k) H_k \nabla \mathcal{L}_k \Big] \Big|_{(s,a)}. \end{aligned} \tag{6.26}$$

If $\alpha_k$ satisfies

$$\Big| 1 - \alpha_k \nabla Q_k^T H_k \nabla Q_k + \frac{\alpha_k^2}{2} \nabla Q_k^T H_k \nabla^2 Q_k H_k \nabla \mathcal{L}_k \Big| < 1, \tag{6.27}$$

then

$$\|Q_{k+1} - Q^*\|_\infty < \|Q_k - Q^*\|_\infty. \tag{6.28}$$

Therefore, $Q_k$ converges to $Q^*$ when $k \to \infty$. Considering our assumptions on the bounds of the eigenvalues of $\nabla^2 Q_k$ and $H_k$, we can derive (6.22) from (6.27). Recursion on (6.26) from $k = 0$ to $k + 1$ results in (6.21). □

### 6.5.3 Computation time

Let us compare the cost of deep L-BFGS Q-learning in Algorithm 15 with DQN algorithm in (Mnih et al., 2015) that uses a variant of SGD. Assume that the cost of computing gradient is $\mathcal{O}(bn)$ where $b$ is the batch size. The real cost is probably less than this due to the parallel computation on GPUs. Let's assume that we run

both algorithm for $L$ steps. We update the weights every $b$ steps. Hence there is $L/b$ maximum updates in our algorithm. The SGD batch size in Mnih et al. (2015), $b_s$, is smaller than $b$, but the frequency of the update is high, $f \ll b$. Each iteration of the L-BFGS algorithm update introduces the cost of computing the gradient, $g_k^{(O_k)}$, which is $\mathcal{O}(bn)$, the cost of computing the search step, $p_k = -H_k g_k^{(O_k)}$, using L-BFGS two-loop recursion (Algorithm 14), which is $\mathcal{O}(4mn)$, and the cost of satisfying the Wolfe conditions (6.9) to find a step size that usually satisfies for $\alpha = 1$ and, in some steps, requires recomputing the gradient $z$ times. Therefore we have

$$\begin{aligned} \frac{\text{Cost of Algorithm 14}}{\text{Cost of DQN (Mnih et al., 2015)}} &= \frac{(L/b)(zbn + 4mn)}{(L/f)(b_s n)} \\ &= \frac{fz}{b_s} + \frac{4fm}{bb_s}. \end{aligned} \tag{6.29}$$

In our algorithm, we use a quite large batch size to compute less noisy gradients. With $b = 2048$, $b_s = 32$, $f = 4$, $z = 5$, $m = 20$, the runtime cost ratio will be around $0.63 < 1$. Although the per-iteration cost of the the SGD algorithm is lower than L-BFGS, the total training time of our algorithm is less than DQN (Mnih et al., 2015) for the same number of RL steps due to the need for less frequent updates in the L-BFGS method.

## 6.6 Experiments on ATARI 2600 Games

We performed experiments using our approach (Algorithm 15) on six ATARI 2600 games – Beam-Rider, Breakout, Enduro, Q*bert, Seaquest, and Space Invaders. We used OpenAI's gym ATARI environments (Brockman et al., 2016) which are wrappers on the Arcade Learning Environment emulator (Bellemare et al., 2013). These games have been used by other researchers investigating different learning methods (Bellemare et al., 2012, 2013; Hausknecht et al., 2014; Mnih et al., 2015; Schulman et al., 2015), and, hence, they serve as benchmark environments for the evaluation of deep reinforcement learning algorithms.

We used DeepMind's Deep Q-Network (DQN) architecture, described in Mnih et al. (2015), as a function approximator for $Q(s, a; w)$. The same architecture was used to train agents to play the different ATARI games. The raw Atari frames, which are $210 \times 160$ pixel images with a 128 color palette, were preprocessed by first converting their RGB representation to gray-scale and then down-sampling the images to be $110 \times 84$ pixels. The final input representation is obtained by cropping an $84 \times 84$ region of the image that roughly captures the playing area. The stack of the last 4 consecutive frames was used to produce the input, of size $(4 \times 84 \times 84)$, to the $Q$-function. The first hidden layer of the network consisted of 32 convolutional filters of size $8 \times 8$ with stride 4, followed by a Rectified Linear Unit (ReLU) for nonlinearity. The second hidden layer consisted of 64 convolutional filters of size $4 \times 4$ with stride 2, followed by a ReLU function. The third layer consisted of 512 fully-connected linear

units, followed by ReLU. The output layer was a fully-connected linear layer with an output, $Q(s, a_i, w)$, for each valid joystick action, $a_i \in \mathcal{A}$. The number of valid joysticks actions, i.e. $|\mathcal{A}|$, was 9 for Beam-Rider, 4 for Breakout, 9 for Enduro, 6 for Q*Bert, 18 for Seaquest, and 6 for Space-Invaders.

We only used 2 million ($2000 \times 1024$) Q-learning training steps for training the network on each game (instead of 50 million steps that was used originally in Mnih et al. (2015)). The training was stopped when the norm of the gradient, $\|g_k\|$, was less than a threshold. We used $\epsilon$-greedy for an exploration strategy, and, similar to Mnih et al. (2015), the exploration rate, $\epsilon$, was annealed linearly from 1 to 0.1.

Every 10,000 steps, the performance of the learning algorithm was tested by freezing the Q-network's parameters. During the test time, we used $\epsilon = 0.05$. The greedy action, $\max_a Q(s, a; w)$, was chosen by the Q-network 95% of the times and there was 5% randomness, similar to the DeepMind implementation in Mnih et al. (2015).

Inspired by Mnih et al. (2015), we also used separate networks to compute the target values, $\mathcal{Y} = r + \gamma \max_{a'} Q(s', a', w_{k-1})$, which was essentially the network with parameters in previous iterate. After each iteration of the multi-batch line search L-BFGS, $w_k$ was updated to $w_{k+1}$, and the target network's parameter $w_{k-1}$ was updated to $w_k$.

Our optimization method was different than DeepMind's RMSProp method, used in Mnih et al. (2015) (which is a variant of SGD). We used a stochastic line search L-BFGS method as the optimization method (Algorithm 15). There are a few important differences between our implementation of deep reinforcement learning and DeepMind's DQN algorithm.

We used a quite large batch size, $b$, in comparison to Mnih et al. (2015). We experimented with different batch sizes $b \in \{512, 1024, 2048, 4096, 8192\}$. The experience memory, $\mathcal{D}$, had a capacity of $b$ also. We used one NVIDIA Tesla K40 GPU with 12GB GDDR5 RAM. The entire experience memory, $\mathcal{D}$, could fit in the GPU RAM with a batch size of $b \leq 8192$.

After every $b$ steps of interaction with the environment, the optimization step in Algorithm 15 was ran. We used the entire experience memory, $\mathcal{D}$, for the overlap, $O_k$, between two consecutive samples, $J_k$ and $J_{k+1}$, to compute the gradient in (6.15) as well as $\mathbf{y}_k$ in (6.13). Although the DQN algorithm used a smaller batch size of 32, the frequency of optimization steps was high (every 4 steps). We hypothesize that using the smaller batch size made the computation of the gradient too noisy, and, also, this approach doesn't save significant computational time, since the overhead of data transfer between GPU and CPU is more costly than the computation of the gradient over a bigger batch size, due to the power of parallelism in a GPU. Once the overlap gradient, $g_k^{(O_k)}$, was computed, we computed the gradient, $g_k^{(J_k)}$, for the current sample, $J_k$, in (6.14) by memorizing and using the gradient information from the previous optimization step. Then, the L-BFGS two loop-recursion in Algorithm 14 was used to compute the search direction $p_k = -H_k g_k^{(J_k)}$.

After finding the quasi-Newton decent direction, $p_k$, the Wolfe Condition (6.9) was applied to compute the step size, $\alpha_k \in [0.1, 1]$, by satisfying the sufficient decrease

and the curvature conditions (Wolfe, 1969; Nocedal and Wright, 2006). During the optimization steps, either the step size of $\alpha_k = 1$ satisfied the Wolfe conditions, or the line search algorithm iteratively used smaller $\alpha_k$ until it satisfied the Wolfe conditions or reached a lower bound of 0.1. The original DQN algorithm used a small fixed learning rate of 0.00025 to avoid the execrable drawback of the noisy stochastic gradient decent step, which makes the learning process very slow.

The vectors $\mathbf{s}_k = w_{k+1} - w_k$ and $\mathbf{y}_k = g_{k+1}^{(O_k)} - g_k^{(O_k)}$ were only added to the recent collections $S_k$ and $Y_k$ if $\mathbf{s}_k^T \mathbf{y}_k > 0$ and not close to zero. We applied this condition to *cautiously* preserve the positive definiteness of the L-BFGS matrices $B_k$. Only the $m$ recent $\{(\mathbf{s}_i, \mathbf{y}_i)\}$ pairs were stored into $S_k$ and $Y_k$ ($|S_k| = m$ and $|Y_k| = m$) and the older pairs were removed from the collections. We experimented with different L-BFGS memory sizes $m \in \{20, 40, 80\}$.

All code is implemented in the Python language using Pytorch, NumPy, and SciPy libraries, and it is available at `http://rafati.net/quasi-newton-rl`.

## 6.7 Results and Discussions

The average of the maximum game scores is reported in Figure 6.1 (a). The error bar in Figure 6.1 (a) is the standard deviation for the simulations with different batch size, $b \in \{512, 1024, 2048, 4096\}$, and different L-BFGS memory size, $m \in \{20, 40, 80\}$, for each ATARI game (total of 12 simulations per each task). All simulations regardless of the batch size, $b$, and the memory size, $m$, exhibited robust learning. The average training time for each task, along with the empirical loss values, $\mathcal{L}(w_k)$, is shown in Figure 6.1 (b).

The Coefficient of Variation (C.V.) for the test scores was about 10% for each ATARI task. (The coefficient of variation is defined as the standard deviation divided by the mean). We did not find a correlation between the test scores and the different batch sizes, $b$, or the different L-BFGS memory sizes, $m$. The coefficient of variation for the training times was 50% for each ATARI task. Hence, we did not find a strong correlation between the training time and the different batch sizes, $b$, or the different L-BFGS memory sizes, $m$. In most of the simulations, the loss for the training time, as shown in Figure 6.1 (b), was very small.

The test scores and the training loss, $\mathcal{L}_k$, for the six ATARI 2600 environments is shown in Figure 6.2 using the batch size of $b = 2048$ and L-BFGS memory size $m = 40$.

The results of the Deep L-BFGS Q-Learning algorithm is summarized in Table 6.1, which also includes an expert human performance and some recent model-free methods: the Sarsa algorithm (Bellemare et al., 2013), the *contingency aware* method from (Bellemare et al., 2012), deep Q-learning (Mnih et al., 2013), and two methods based on policy optimization called Trust Region Policy Optimization (TRPO vine and TRPO single path) (Schulman et al., 2015) and the Q-learning with the SGD method. Our method outperformed most other methods in the Space Invaders game. Our deep L-BFGS Q-learning method consistently achieved reasonable scores in the

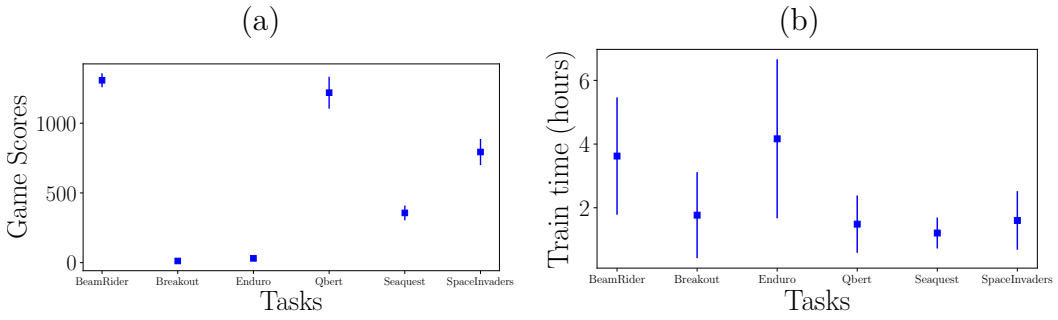



Figure 6.1: (a) Test scores (b) Total training time for ATARI games.

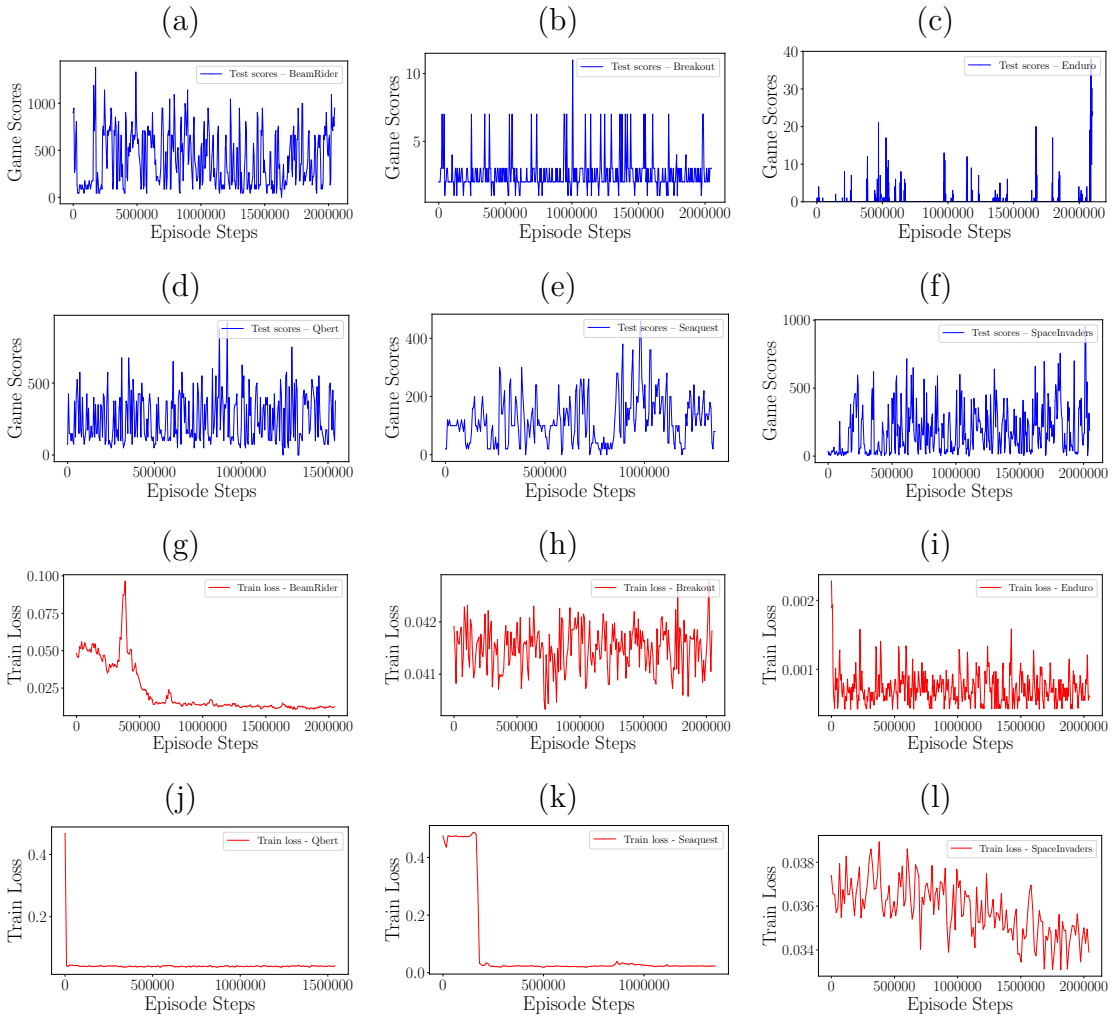



Figure 6.2: (a) – (f) Test scores and (g) – (l) training loss for six ATARI games — Beam Rider, Breakout, Enduro, Q*bert, Seaquest, and Space Invaders. The results are form simulations with batch size $b = 2048$ and the L-BFGS memory size $m = 40$.

other games. Our simulations were only trained on about 2 million Q-learning steps (much less than other methods). DeepMind DQN method outperformed our algorithm on most of the games, except on the Space-Invaders game.

Table 6.1: Best Game Scores for ATARI 2600 Games with different learning methods. Beam Rider (BR), Breakout (BO), Enduro (EO), Q*bert (Q*B), Seaquest (SQ), and Space Invaders (SI)

| **Method** | **BR** | **BO** | **EO** | **Q*B** | **SQ** | **SI** |
|---|---|---|---|---|---|---|
| Random | 354 | 1.2 | 0 | 157 | 110 | 179 |
| Human | 7456 | 31 | 368 | 18900 | 28010 | 3690 |
| Sarsa (Bellemare et al., 2013) | 996 | 5.2 | 129 | 614 | 665 | 271 |
| Contingency (Bellemare et al., 2012) | 1743 | 6 | 159 | 960 | 723 | 268 |
| HNeat Pixel (Hausknecht et al., 2014) | 1332 | 4 | 91 | 1325 | 800 | 1145 |
| DQN (Mnih et al., 2013) | 4092 | 168 | 470 | 1952 | 1705 | 581 |
| TRPO, Single path (Schulman et al., 2015) | 1425 | 10 | 534 | 1973 | 1908 | 568 |
| TRPO, Vine (Schulman et al., 2015) | 859 | 34 | 431 | 7732 | 7788 | 450 |
| SGD ($\alpha = 0.01$) | 804 | 13 | 2 | 1325 | 420 | 735 |
| SGD ($\alpha = 0.00001$) | 1092 | 14 | 1 | 1300 | 380 | 975 |
| Our method | 1380 | 18 | 49 | 1525 | 600 | 955 |

The training time for our simulations were on the order of 3 hours (4 hours for Beam-Rider, 2 hours for Breakout, 4 hours for Enduro, 2 hours for Q*bert, 1 hour for Seaquest, and 2 hours for Space-Invaders). Our method outperformed all other methods on the computational time. For example, 500 iterations of the TRPO algorithm took about 30 hours (Schulman et al., 2015). We also compared our method with the SGD method. For each task, we trained the Q-learning algorithm using the SGD optimization method for two million Q-learning training steps. We examined two different learning rates: a relatively large learning rate, $\alpha = 0.01$, and a very small learning rate, $\alpha = 0.00001$. The other parameters were adopted from Mnih et al. (2015). The game scores with our method outperformed the SGD method in most of the simulations (11 out of 12 times) (see Table 6.1). Although the computation time per iteration of the SGD update is lower than our method, but the total training time of the SGD method is much slower than our method due to the higher frequency of the parameter updates in the SGD method as opposed to our L-BFGS line-search method. See Table 6.2 for the results of the training time for each task, using different optimization methods, L-BFGS and SGD.

Table 6.2: Average training time for ATARI 2600 games with different learning methods (in hours). Beam Rider (BR), Breakout (BO), Enduro (EO), Q*bert (Q*B), Seaquest (SQ), and Space Invaders (SI)

| **Method** | **BR** | **BO** | **EO** | **Q*B** | **SQ** | **SI** |
|---|---|---|---|---|---|---|
| SGD ($\alpha = 0.01$) | 4 | 2 | 8 | 8 | 8 | 1 |
| SGD ($\alpha = 0.00001$) | 7 | 11 | 7 | 6 | 8 | 6 |
| Our method | 4 | 2 | 4 | 2 | 1 | 1 |

## 6.8 Future Work

In future work, we will consider optimization methods based on trust-regions. As we showed in Chapter 5, trust-region methods require choosing some hyperparameters, like proper initial trust region radius, but they might converge faster than line search since they do not require satisfying the curvature condition, and the sufficient decrease condition. Additionally, trust-region algorithms can shrink or expand the trust-region radius based on the quality of the search step. Also, since the true Hessian is indefinite, using indefinite quasi-Newton matrices, like Symmetric Rank 1 (SR1), or Full Broyden Class (FBC) within trust-region methods might lead to better convergence properties. We will study these methods as future work.

We also intend to study Hessian-free optimization methods for model-free RL within conjugate gradient framework. Conjugate gradient is a method for minimizing the quadratic model of the objective function and solving large linear systems of equations. Newton's method is an algorithm for solving nonlinear equations, and can be used as a second-order optimization method within the conjugate gradient framework. In Newton's method, we need to compute the Hessian matrix, $H = \nabla^2 \mathcal{L}(w)$, to find the next iterate, $w_{k+1} = w_k - H_k^{-1} \nabla \mathcal{L}(w_k)$. Although finding the Hessian and its inverse in large-scale problems is not possible, we can compute the Hessian times any vector with just two computations of the gradient

$$Hp = \frac{\nabla \mathcal{L}(w + \epsilon p) - \nabla \mathcal{L}(w)}{\epsilon} + O(\epsilon), \tag{6.30}$$

where $\epsilon$ is small. Also, for the specific case of neural networks, we can compute $Hp$ directly using an algorithm similar to normal backpropagation (Pearlmutter, 1994). Hence, we can employ conjugate gradient methods to find the next iteration $w_{k+1}$ (Le et al., 2011a). See Nocedal and Wright (2006) for details on conjugate gradient methods. The Newton's methods have quadratic rates of convergence, which is theoretically better than the quasi-Newton methods, with superlinear convergence rates.

## 6.9 Conclusions

We proposed and implemented a novel optimization method based on line search limited-memory BFGS for the deep reinforcement learning framework. We tested our method on six classic ATARI 2600 games. The L-BFGS method attempts to approximate the Hessian matrix by constructing positive definite matrices with low-rank updates. Due to the nonconvex and nonlinear loss functions arising in deep reinforcement learning, our numerical experiments show that using the curvature information in computing the search direction leads to a more robust convergence when compared to the SGD results. Our proposed deep L-BFGS Q-Learning method is designed to be efficient for parallel computations on GPUs. Our method is much faster than the existing methods in the literature, and it is memory efficient since it does not need to store a large experience replay memory.

# Chapter 7

# Concluding Remarks

Reinforcement Learning (RL) algorithms allow artificial agents to improve their selection of actions so as to increase rewarding experiences in their environments. These algorithms typically learn a mapping from the agent's current sensed state to a selected action. In the model-free RL algorithms, like Temporal Difference (TD) Learning, the agent learns the value of taking an action at any state, without learning a model of the environment (i.e. the state-transition probabilities and the reward function). TD algorithms, such as SARSA, and Q-learning have been very successful on a broad range of control tasks. But learning can become intractably slow as the state space of the environment grows. This has motivated methods that learn internal representations of the agent's state, effectively reducing the size of the state space and restructuring state representations in order to support generalization to novel situations.

In this dissertation, I have investigated biologically inspired techniques for learning useful representations for model-free reinforcement learning, as well as numerical optimization methods for improving learning. There are three parts to this investigation: (1) learning sparse representations in reinforcement learning, (2) learning representations in model-free hierarchical reinforcement learning, and (3) optimization methods in reinforcement learning. These three contributions provide a foundation for scaling reinforcement learning to more complex control problems through the learning of improved internal representations.

Many machine learning tasks (including RL) require solving empirical risk minimization problems which are highly nonconvex and nonlinear. In the context of large-scale machine learning, and large-scale nonconvex optimization, I have contributed to the deep learning literature by implementing novel trust-region quasi-Newton methods. The proposed methods do not require hyperparamter tuning, and they have robust convergence with preferred generalization characteristics.

## 7.1 Summary of Contributions

### Learning Sparse Representations in RL

Using a function approximator to learn the value function (an estimate of the expected future reward from a given environmental state) has the benefit of supporting generalization across similar states, but this approach can produce a form of catastrophic interference that hinders learning, mostly arising in cases in which similar states have widely different values. Computational neuroscience models have shown that a combination of feedforward and feedback inhibition in neural circuits naturally produces sparse conjunctive codes over a collection of excitatory neurons (Noelle, 2008).

I have implemented efficient algorithms that incorporate a kind of lateral inhibition into artificial neural network layers, driving these machine learning systems to produce sparse conjunctive internal representations. I have shown how such learned representations lead to successful learning of control tasks that had previously been seen as problematic. I have produced computational simulation results as preliminary evidence that learning such sparse representations of the state of an RL agent can help compensate for weaknesses of artificial neural networks in TD Learning. See Chapter 3 or our published papers, Rafati and Noelle (2015, 2017), for more details.

### Learning Representations in Model-Free HRL

Hierarchical Reinforcement Learning (HRL) methods attempt to address the scalability issues of RL by learning policies at multiple levels of temporal abstraction. Abstraction can be had by subgoal discovery, in concert with the learning of corresponding skill policies to achieve those subgoals. Many approaches to subgoal discovery in HRL depend on the analysis of a model of the environment, but the need to learn such a model introduces its own problems of scale.

I have implemented a novel model-free method for subgoal discovery using incremental unsupervised learning over a small memory of the most recent experiences of the agent. When combined with an intrinsic motivation learning mechanism, and temporal abstraction, this method learns subgoals and skills together, based on experiences in the environment. Thus, I have produced an original approach to HRL that does not require the acquisition of a model of the environment, suitable for large-scale applications. See Chapter 4, or our published papers, Rafati and Noelle (2019a,b,c,d), for more details.

In search of further biological inspiration, I have attempted to find correlates of the components of the unified model-free HRL framework in regions of the human brain. I have found associations to the prefrontal cortex, motor areas, the basal ganglia, the dopamine system, and the hippocampus. I hope that this work will lead to new ideas about the functions of major parts of the brain. In future work, I hope to combine the model-free HRL framework with model-based methods to incorporate planning into the learning of the representations. I hypothesize that this unified HRL

framework will be able to solve much more complex tasks. Finally, I am interested in understanding the connections between the components of the HRL framework with Heideggerian phenomenology (Yoshimi, 2017).

## Quasi-Newton Optimization in Large-scale Machine Learning

Intelligent processing of complex signals, such as in image recognition and deep reinforcement learning is often performed by a parameterized hierarchy of nonlinear layers, as in deep neural networks. Deep learning and Deep RL algorithms often require solving a highly nonlinear and nonconvex unconstrained optimization problem.

Methods for solving such optimization problems in deep learning and deep RL have generally been restricted to the class of first-order algorithms, like stochastic gradient descent (SGD). The major drawback of the SGD methods is that they have the undesirable effect of not escaping saddle points in the objective function. Furthermore, these methods typically require exhaustive trial and error to fine tune many learning parameters. Since the stochastic gradient is noisy, the learning rate (step size) is generally chosen to be very small, causing each experience to have only a small effect. Hence, in order to succeed at optimization, SGD methods must repeatedly replay a large number of recorded experiences. This is computationally expensive, and it requires a large amount of memory.

Using second derivative information can result in improved convergence properties, but computing the Hessian matrix for large-scale problems is not practical. Quasi-Newton methods require only first-order gradient information. By incorporating the curvature information into the search direction using a low rank approximation of the Hessian, quasi-Newton methods can result in a superlinear convergence, which makes them attractive alternatives to SGD methods. The limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) approach is one of the most popular quasi-Newton methods, relying on the construction of positive definite Hessian approximations.

Quasi-Newton methods are another first-order optimization methods which can result in a superlinear convergence. They can construct a low rank approximation of the Hessian and incorporate the curvature information into the search direction, using only first-order gradients. The limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) approach is one of the most popular quasi-Newton methods, relying on the construction of positive definite Hessian approximations.

I have implemented efficient algorithms based on L-BFGS optimization method suitable for deep learning an deep RL applications to improve the quality of representation learning, as well as convergence properties.

## Trust-region L-BFGS Optimization in Deep Learning

There are two strategies for quasi-Newton optimizations, *line-search* and the *trust-region* method. Both methods seek to minimize a quadratic model of the objective function around the current iterate. The line-search strategy finds a descent direction

and then imposes a sufficient decrease and curvature condition to find the proper step length along the search direction. The trust-region method requires solving a constrained optimization subproblem, which is more complicated, but there are efficient methods to find a global optimizer of the quadratic model in the trust-region.

I have implemented L-BFGS optimization under both *trust-region* and *line-search* frameworks, and I have produced evidence that this approach is efficient for deep learning problems such as classification and regression from big data. The results show that the trust-region approach converges much faster than the line-search method, and, when the batch size is decreased, the trust-region method outperforms the line-search method in terms of learned performance. See Chapter 5 or our published paper, Rafati et al. (2018), for more details.

## Improving L-BFGS Initialization for Trust-Region Methods

Since the true Hessian matrix is not necessarily positive definite, an extra initialization condition is required to be introduced when constructing the L-BFGS matrices to avoid false negative curvature information. I have explored three initialization methods for the L-BFGS matrices, within a trust-region framework. I have provided empirical results on a benchmark classification task (the MNIST digits dataset) to compare the performance of the trust-region algorithm with different L-BFGS initialization methods. See Chapter 5 or our published paper, Rafati and Marcia (2018), for more details.

## Quasi-Newton Optimization Methods in Deep RL

I have contributed to the design of an efficient optimization method, based on the L-BFGS quasi-Newton method within line search strategy, offering it as an alternative to SGD methods. This new method bridges the disparity between first-order methods and second-order methods by continuing to use gradient information to calculate low-rank Hessian approximations. I have produced empirical results applying this method to RL agents learning a subset of the classic ATARI 2600 games. The results show a robust convergence with preferred generalization characteristics, as well as fast training time and no need for an experience replay mechanism. I have also conducted a formal analysis on the effect of different optimization methods on the performance of deep RL algorithms, and I have assessed the resulting quality of learned representations. See Chapter 6 or our unpublished manuscript, Rafati and Marcia (2019), for more details.

## 7.2 Future Work

Representation learning in machine learning, deep learning, and reinforcement learning is an open problem. In this dissertation, we focused on a small subset of methods for learning representations within temporal difference learning framework, a model-free reinforcement learning. The proposed methods can be modified and used within model-based reinforcement learning. In model-based approach, the agent can incorporate *planning* into the *learning* process by learning a model of the environment. In future work, I will study methods for learning representations of the agent's state within the model-based reinforcement learning framework.

Proofs on effectiveness of the proposed methods in this dissertation (as well as majority of the deep learning and the deep RL literature) rely on the empirical results on some numerical simulations, which are very time consuming. In the future, I will attempt to produce formal convergence analyses in order to compute upper-bounds and lower-bounds of each proposed method in order to analyze the limits and power of each method.

Model selection is another major open problem in machine learning and reinforcement learning that I have not addressed in this dissertation. In future work, I will study different families of function approximators for the value function and investigate their effects on learning representations of the agent's state.

The proposed methods of this dissertation can be applied to a set of real-world applications, such as autonomous driving, robotics, economics, and any other large-scale applications that require learning optimal decision making policies to maximize a profit and minimize a loss. In the future, I will investigate the effectiveness of the proposed methods on real-world applications.

# Bibliography


Adhikari, L., DeGuchy, O., Erway, J. B., Lockhart, S., and Marcia, R. F. (2017). Limited-memory trust-region methods for sparse relaxation. In *Wavelets and Sparsity XVII*, volume 10394. International Society for Optics and Photonics.

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller CMAC. *Journal of Dynamic Systems, Measurement, and Control*, 97(3):220–227.

Ammar, H. B., Tuyls, K., Taylor, M. E., Driessens, K., and Weiss, G. (2012). Reinforcement learning transfer via sparse coding. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '12.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38.

Bacon, P., Harb, J., and Precup, D. (2017). The option-critic architecture. In *AAAI*.

Badre, D., Kayser, A., and D'Esposito (2010). Frontal cortex and the discovery of abstract action rules. *Neuron*, 66:315–326.

Bakker, B. and Schmidhuber, J. (2004). Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proceedings of the 8-th Conference on Intelligent Autonomous Systems, IAS-8*, pages 438–445.

Barto, A. G. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1):41–77.

Bellemare, M., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 1471–1479. Curran Associates, Inc.

Bellemare, M. G., Veness, J., and Bowling, M. H. (2012). Investigating contingency awareness using atari 2600 games. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Berahas, A. S., Nocedal, J., and Takac, M. (2016). A multi-batch L-BFGS method for machine learning. In *Advances in Neural Information Processing Systems 29*, pages 1055–1063.

Bertsekas, D. P. (2000). *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition.

Bollapragada, R., Byrd, R., and Nocedal, J. (2016). Exact and inexact subsampled non methods for optimization. *ArXiv e-prints*.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

Bottou, L., Curtis, F., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311.

Botvinick, M. and Weinstein, A. (2014). Model-based hierarchical reinforcement learning and human action control. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 369.

Botvinick, M. M., Niv, Y., and Barto, A. C. (2009). Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *Cognition*, 113(3):262–280.

Boyan, J. A. and Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA. MIT Press.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym.

Broyden, C. G. (1970). The convergence of a class of double-rank minimization algorithms 1. general considerations. *SIAM Journal of Applied Mathematics*, 6(1):76–90.

Brust, J., Burdakov, O., Erway, J. B., and Marcia, R. F. (2017a). Dense initializations for limited-memory quasi-newton methods. *ArXiv e-prints*.

Brust, J., Erway, J. B., and Marcia, R. F. (2017b). On solving L-SR1 trust-region subproblems. *Computational Optimization and Applications*, 66(2):245–266.

Burdakov, O., Gong, L., Yuan, Y.-X., and Zikrin, S. (2016). On efficiently combining limited memory and trust-region techniques. *Mathematical Programming Computation*, 9:101–134.

Byrd, R. H., Hansen, S. L., Nocedal, J., and Singer, Y. (2016). A stochastic quasi-newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031.

Byrd, R. H., Nocedal, J., and Schnabel, R. B. (1994). Representations of quasi-Newton matrices and their use in limited-memory methods. *Math. Program.*, 63:129–156.

Chalmers, E., Luczak, A., and Gruber, A. J. (2016). Computational properties of the hippocampus increase the efficiency of goal-directed foraging through hierarchical reinforcement learning. *Frontiers in Computational Neuroscience*, 10.

Conn, A. R., Gould, N. I. M., and Toint, P. L. (2000). *Trust-Region Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Dayan, P. (1992). The convergence of TD($\lambda$) for general $\lambda$. *Machine Learning*, 8:341–362.

Dayan, P. and Hinton, G. E. (1992). Feudal reinforcement learning. In *NeurIPS*.

Dayan, P. and Niv, Y. (2008). Reinforcement learning: The good, the bad and the ugly. *Current Opinion in Neurobiology*, 18:185–196.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Diuk, C., Schapiro, A., Córdova, N., Ribas-Fernandes, J., Niv, Y., and Botvinick, M. (2013). Divide and conquer: hierarchical reinforcement learning and task decomposition in humans. In *Computational and robotic models of the hierarchical organization of behavior*, pages 271–291. Springer.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. (2019). Go-explore: a new approach for hard-exploration problems. *ArXiv e-prints arXiv:1901.10995*.

Erway, J. B., Griffin, J., Marcia, R. F., and Omheni, R. (2018). Trust-region algorithms for training responses: Machine learning methods using indefinite hessian approximations. *ArXiv e-prints*.

Fletcher, R. (1970). A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322.

Fox, R., Krishnan, S., Stoica, I., and Goldberg, K. Y. (2017). Multi-level discovery of deep options. *arXiv preprint arXiv:1703.08294*.

Fragkiadaki, K., Arbelaez, P., Felsen, P., and Malik, J. (2015). Learning to segment moving objects in videos. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4083–4090.

French, R. M. (1991). Using semi-distributed representations to overcome catastrophic forgetting in connectionist networks. In *Proceedings of the 13th Annual Cognitive Science Society Conference*, pages 173–178, Hillsdale, NJ. Lawrence Erlbaum.

Gay, D. M. (1981). Computing optimal locally constrained steps. *SIAM Journal on Scientific and Statistical Computing*, 2(2):186–197.

Goel, S. and Huber, M. (2003). Subgoal discovery for hierarchical reinforcement learning using learned policies. In *FLAIRS Conference*, pages 346–350. AAAI Press.

Goldfarb, D. (1970). A family of variable-metric methods derived by variational means. *Mathematics of computation*, 24(109):23–26.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition.

Hausknecht, M., Lehman, J., Miikkulainen, R., and Stone, P. (2014). A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366.

Hengst, B. (2010). *Hierarchical Reinforcement Learning*, pages 495–502. Springer US, Boston, MA.

Hodge, V. J. and Austin, J. (2004). A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126.

Jaakkola, T., Jordan, M. I., and Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201.

Kandel, E., Schwartz, J., Jessell, T., Siegelbaum, S., and Hudspeth, A. J. (2012). *Principles of Neural Science*. McGraw-Hill, New York, fifth edit edition.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krishnamurthy, R., Lakshminarayanan, A. S., Kumar, P., and Ravindran, B. (2016). Hierarchical reinforcement learning using spatio-temporal abstractions and deep neural networks. *ArXiv e-prints (arxiv:1605.05359)*.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. (2016). Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 3675–3683.

Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., and Ng, A. Y. (2011a). On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272.

Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., and Ng, A. Y. (2011b). On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272. Omnipress.

LeCun, Y. (1998). The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

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

LeCun, Y. and Others (2015). Lenet5, convolutional neural networks. page 20.

Levy, K. Y. and Shimkin, N. (2011). Unified inter and intra options learning using policy gradient methods. In *European Workshop on Reinforcement Learning*. Springer.

Li, Z., Narayan, A., and Leong, T.-Y. (2017). An efficient approach to model-based hierarchical reinforcement learning.

Liu, D. C. and Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528.

Liu, V., Kumaraswamy, R., Le, L., and White, M. (2018). The utility of sparse representations for control in reinforcement learning. *arXiv e-prints (1811.06626)*.

Lyu, D., Yang, F., Liu, B., and Gustafson, S. (2019). SDRL: Interpretable and Data-efficient Deep Reinforcement Learning Leveraging Symbolic Planning. In *33rd AAAI Conference on Artificial Intelligence (AAAI-19), Honolulu, HI, USA*.

Machado, M. C., Bellemare, M. G., and Bowling, M. H. (2017). A Laplacian Framework for Option Discovery in Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, {ICML} 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 2295–2304.

Machado, M. C. and Bowling, M. (2016). Learning purposeful behaviour in the absence of rewards.

Maillard, O., Ryabko, D., and Munos, R. (2011). Selecting the state-representation in reinforcement learning. In *Advances in Neural Information Processing Systems 24*, pages 2627–2635. Curran Associates, Inc.

Mannor, S., Menache, I., Hoze, A., and Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04.

Martens, J. (2010). Deep learning via Hessian-free optimization. In *Proc. of the 27th Intl.˜Conf.˜on Machine Learning (ICML)*, pages 735–742.

Martens, J. and Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference of on Machine Learning (ICML11)*, pages 1033–1040.

Martens, J. and Sutskever, I. (2012). Training deep and recurrent networks with hessian-free optimization. In *Neural Networks: Tricks of the Trade*, pages 479–535. Springer.

McGovern, A. and Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 361–368.

Melo, F. S., Meyn, S. P., and Ribeiro, M. I. (2008). An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th International Conference on Machine Learning*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *ArXiv e-prints (arxiv:1312.5602)*.

Mnih, V., Kavukcuoglu, K., Silver, D., and Others (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Montague, P. R., Dayan, P., and Sejnowski, T. J. (1996). A framework for mesencephalic dopamine systems based on predictive hebbian learning. *Journal of Neuroscience*, 16:1936–1947.

Moré, J. J. and Sorensen, D. C. (1983). Computing a trust region step. *SIAM Journal on Scientific and Statistical Computing*, 4(3):553–572.

Nesterov, Y. (2013). *Introductory Lectures on Convex Optimization: A Basic Course*. Springer Science & Business Media.

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, New York, 2nd edition.

Noelle, D. C. (2008). Function follows form: Biologically guided functional decomposition of memory systems. In *Biologically Inspired Cognitive Architectures — Papers from the 2008 AAAI Fall Symposium*.

O'Reilly, R. C. (2001). Generaliztion in interactive networks: The benefits of inhibitory competition and Hessian learning. *Neural Computation*, 13:1199–1242.

O'Reilly, R. C. and McClelland, J. L. (1994). Hippocampal conjunctive encoding, storage, and recall: Avoiding a trade-off. *Hippocampus*, 4(6):661–682.

O'Reilly, R. C. and Munakata, Y. (2001). *Computational Explorations in Cognitive Neuroscience*. MIT Press, Cambridge, Massachusetts.

Parr, R. and Russell, S. J. (1997). Reinforcement learning with hierarchies of machines. In *NeurIPS*.

Pearlmutter, B. A. (1994). Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160.

Pickett, M. and Barto, A. G. (2002). Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 506–513.

Preston, A. R. and Eichenbaum, H. (2013). Interplay of hippocampus and prefrontal cortex in memory. *Current Biology*, 23(17):764–773.

Rafati, J., DeGuchy, O., and Marcia, R. F. (2018). Trust-region minimization algorithm for training responses (TRMinATR): The rise of machine learning techniques. In *26th European Signal Processing Conference (EUSIPCO 2018), Rome, Italy*.

Rafati, J. and Marcia, R. F. (2018). Improving L-BFGS initialization for trust-region methods in deep learning. In *17th IEEE International Conference on Machine Learning and Applications, Orlando, Florida*.

Rafati, J. and Marcia, R. F. (2019). Deep reinforcement learning via L-BFGS optimization. *arXiv e-print (arXiv:1811.02693)*.

Rafati, J. and Noelle, D. C. (2015). Lateral inhibition overcomes limits of temporal difference learning. In *Proceedings of the 37th Annual Cognitive Science Society Meeting, Pasadena, CA, USA*.

Rafati, J. and Noelle, D. C. (2017). Sparse coding of learned state representations in reinforcement learning. In *Conference on Cognitive Computational Neuroscience, New York City, NY, USA*.

Rafati, J. and Noelle, D. C. (2019a). Learning representations in model-free hierarchical reinforcement learning. In *33rd AAAI Conference on Artificial Intelligence (AAAI-19), Honolulu, HI, USA.*

Rafati, J. and Noelle, D. C. (2019b). Learning representations in model-free hierarchical reinforcement learning. *arXiv e-print (arXiv:1810.10096).*

Rafati, J. and Noelle, D. C. (2019c). Unsupervised methods for subgoal discovery during intrinsic motivation in model-free hierarchical reinforcement learning. In *33rd AAAI Conference on Artificial Intelligence (AAAI-19). Workshop on Knowledge Extraction From Games. Honolulu, HI, USA.*

Rafati, J. and Noelle, D. C. (2019d). Unsupervised subgoal discovery method for learning hierarchical representations. In *7th International Conference on Learning Representations, ICLR 2019 Workshop on "Structure & Priors in Reinforcement Learning", New Orleans, Louisiana.*

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.

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

Schultz, W., Apicella, P., and Ljungberg, T. (1993). Responses of monkey dopamine neurons to reward and conditioned stimuli during successive steps of learning a delayed response task. *Journal of Neuroscience*, 13:900–913.

Schultz, W., Dayan, P., and Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599.

Shanno, D. F. (1970). Conditioning of quasi-Newton methods for function minimization. *Mathematics of computation*, 24(111):647–656.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Simsek, Ö., Algorta, S., and Kothiyal, A. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 816–823.

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

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339.

Stolle, M. and Precup, D. (2002). Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pages 212–223.

Strange, B. A., Witter, M. P., Lein, E. S., and Moser, E. I. (2014). Functional organization of the hippocampal longitudinal axis. *Nature Reviews Neuroscience*, 15(10):655–669.

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

Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1st edition.

Sutton, R. S. and Barto, A. G. (2017). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 2nd edition.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211.

Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *JMLR*, 10(Jul):1633–1685.

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

Thrun, S. and Schwartz, A. (1995). Finding structure in reinforcement learning. In *Advances in Neural Information Processing Systems 7*, pages 385–392. MIT Press.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. (2017). Feudal networks for hierarchical reinforcement learning. In *Proceedings of Thirty-fourth International Conference on Machine Learning (ICML-17)*.

Vigorito, C. M. and Barto, A. G. (2010). Intrinsically motivated hierarchical skill learning in structured environments. *IEEE Transactions on Autonomous Mental Development*, 2(2):132–143.

Wah, J. L. and Chuei, Y. C. (2013). A class of diagonal preconditioners for limited memory BFGS method. *Optimization Methods and Software*, 28(2):379–392.

Wolfe, P. (1969). Convergence conditions for ascent methods. *SIAM Review*, 11(2):226–235.

Xu, P., Roosta-Khorasan, F., and Mahoney, M. (2017). Second-order optimization for non-convex machine learning: An empirical study. *ArXiv e-prints*.

Yoshimi, J. (2017). Modeling consciousness using cognitive maps. *Mind and Matter*, 15(1):29–47.

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *ArXiv e-prints (arxiv:1212.5701)*.

Zhang, Z., Xu, Y., Yang, J., Li, X., and Zhang, D. (2015). A survey of sparse representation: Algorithms and applications. *IEEE Access*, 3:490–530.