Efficient Deep Reinforcement Learning via Planning, Generalization, and Improved Exploration

by

Junhyuk Oh

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2018

Junhyuk Oh

junhyuk@umich.edu

ORCID iD: 0000-0003-4383-6396

ACKNOWLEDGEMENTS

First of all, I would like to thank my co-advisors Honglak Lee and Satinder Singh. It has been a great honor to have both of them as my mentors. All of the work presented in this thesis would have not been possible without their advice and valuable insights. They have been not only excellent advisors in research but also very supportive of my academic career. Working with Honglak and Satinder as a PhD student was one of the best decisions that I have ever made.

I would also like to thank Richard Lewis and Jia Deng for serving on my thesis committee and providing constructive feedback on the thesis. It was also a great pleasure to collaborate with Richard Lewis for my early work.

I am thankful for my former and current lab colleagues: Kihyuk Sohn, Scott Reed, Ruben Villigas, Xinchen Yan, Ye Liu, Yuting Zhang, Seunghoon Hong, Lajanugen Logeswaran, Kibok Lee, Sungryull Sohn, Jongwook Choi, Yijie Guo, Xiaoxiao Guo, Nan Jiang, Zeyu Zheng, Vivek Veeriah, John Holler, Qi Zhang, Shun Zhang, Janarthanan Rajendran, Ethan Brooks, Christopher Grimm, Max Smith, Aditya Modi, and Valliappa Chockalingam. It has been a wonderful experience to share joys and hardships with you.

I am thankful to David Silver for hosting me as an intern at DeepMind where I learned lots of insights from David and truly enjoyed discussions and conversations with Arthur Guez, Hado van Hasselt, Tom Schaul, André Barreto, Joseph Modayil, Thomas Degris, Matteo Hessel, Dan Horgan, Simon Osindero, Sasha Vezhnevets, Vlad Mnih, Nando de Freitas, Daan Wierstra, and Razvan Pascanu. I would like to thank Pushmeet Kohli for hosting me as an intern at Microsoft Research and making a lot of connections to other researchers. I would like to thank Pieter Abbeel, Marc Bellemare, Hugo Larochelle, and John Schulman for giving me an opportunity to present my work at their institutions; Doina Precup, Abdel-Rahman Mohamed, Kyunghyun Cho, Shane Gu, and Junyoung Chung for their advice on my academic career; Youngjae Kim and Jaeho Im for being long-term friends in Ann Arbor.

Finally, this thesis is dedicated to my parents and my wife for their love and support. Especially, I would like to thank my wife, Sona Jeong, for being always very supportive and bringing happiness to my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

Reinforcement learning (RL) is a general-purpose machine learning framework, which considers an agent that makes sequential decisions in an environment to maximize its reward. Deep reinforcement learning (DRL) approaches use deep neural networks as non-linear function approximators that parameterize policies or value functions directly from raw observations in RL. Although DRL approaches have been shown to be successful on many challenging RL benchmarks, much of the prior work has mainly focused on learning a single task in a model-free setting, which is often sample-inefficient. On the other hand, humans have abilities to acquire knowledge by learning a model of the world in an unsupervised fashion, use such knowledge to plan ahead for decision making, transfer knowledge between many tasks, and generalize to previously unseen circumstances from the pre-learned knowledge. Developing such abilities are some of the fundamental challenges for building RL agents that can learn as efficiently as humans.

As a step towards developing the aforementioned capabilities in RL, this thesis develops new DRL techniques to address three important challenges in RL: 1) planning via prediction, 2) rapidly generalizing to new environments and tasks, and 3) efficient exploration in complex environments.

The first part of the thesis discusses how to learn a dynamics model of the environment using deep neural networks and how to use such a model for planning in complex domains where observations are high-dimensional. Specifically, we present neural network architectures for action-conditional video prediction and demonstrate improved exploration in RL. In addition, we present a neural network architecture that performs lookahead planning by predicting the future only in terms of rewards and values without predicting observations. We then discuss why this approach is beneficial compared to conventional model-based planning approaches.

The second part of the thesis considers generalization to unseen environments and tasks. We first introduce a set of cognitive tasks in a 3D environment and present memory-based DRL architectures that generalize better to previously unseen 3D environments compared to existing baselines. In addition, we introduce a new multi-task RL problem where the agent should learn to execute different tasks depending on given instructions and generalize to new instructions in a zero-shot fashion. We present a new hierarchical DRL architecture that learns to generalize over previously unseen task descriptions with minimal prior knowledge.

The third part of the thesis discusses how exploiting past experiences can indirectly drive deep exploration and improve sample-efficiency. In particular, we propose a new off-policy learning algorithm, called self-imitation learning, which learns a policy to reproduce past good experiences. We empirically show that self-imitation learning indirectly encourages the agent to explore reasonably good state spaces and thus significantly improves sample-efficiency on RL domains where exploration is challenging.

Overall, the main contribution of this thesis are to explore several fundamental challenges in RL in the context of DRL and develop new DRL architectures and algorithms to address such challenges. This allows us to understand how deep learning can be used to improve sample efficiency, and thus come closer to human-like learning abilities.

CHAPTER I

Introduction

Reinforcement learning (RL) is an area of machine learning that considers an agent which learns to make sequential decisions by interacting with an environment (Sutton and Barto, 1998). At each time-step, the agent chooses an action in the environment and receives a new observation and a reward. The typical goal for the agent agent is to learn a policy that maximizes cumulative reward. RL is a general-purpose learning framework which can address many important aspects of artificial intelligence (AI) and has many applications such as robot control (Kober et al., 2013), recommendation system (Li et al., 2010), and dialog system (Singh et al., 2002).

Deep learning is another area of machine learning that aims to learn hierarchical representations (i.e., abstractions) from raw data (LeCun et al., 2015; Lee, 2010; Hinton and Salakhutdinov, 2006). Deep learning approaches remove the necessity for hand-engineered features which require domain-specific knowledge. Due to the recent advances in hardware, large-scale datasets, optimization methods, and regularization methods, deep neural networks have been successfully applied to many supervised learning problems such as visual recognition (Krizhevsky et al., 2012; Szegedy et al., 2015; Girshick et al., 2014), speech recognition (Hinton et al., 2012), and natural language processing (Mikolov et al., 2013; Cho et al., 2014).

More recently, the success of deep learning has been extended to RL, which created a new research area called *deep reinforcement learning* (DRL). The main idea is to use a deep neural network as a non-linear function approximator for representing a value function or a policy directly from raw observations (e.g., pixel images). By learning from raw observations using neural networks, DRL approaches can learn state representations that are useful for control without requiring any domain knowledge. This approach has turned out to be very successful on challenging RL benchmarks (Bellemare et al., 2013). For example, Mnih et al. (2015) showed that RL agents parameterized by deep neural networks can achieve human-level performance on challenging Atari games (Bellemare et al., 2013) without any game-specific knowledge. Silver et al. (2016, 2017a) showed that a Monte-Carlo Tree Search (MCTS) (Browne et al., 2012) augmented by a deep neural network, which is learned purely from self-play, can beat the best professional Go players in the world.

Although these advances in DRL are remarkable, prior work on DRL has mainly focused on learning a single task in a model-free setting, which is often sample-inefficient. On the other hand, humans have abilities to acquire knowledge by learning a model of the world in an unsupervised fashion, use such knowledge to plan ahead for decision making, transfer knowledge between many tasks, and generalize to unseen circumstances from the pre-learned knowledge. Developing such abilities are some of the fundamental challenges for building RL agents that can learn as efficiently as humans, which has not been much discussed in the DRL area.

The main goal of this thesis is to build more efficient RL agents by developing such abilities through DRL techniques. Specifically, we consider and address three important problems in RL: 1) planning via prediction, 2) generalizing to new environments and tasks, 3) and exploring efficiently in complex environments. We discuss more details below.

Firstly, the ability to predict the future is one of the key aspects of AI, because predicting what would happen in the future amounts to learning and understanding the dynamics of the environment (e.g., physics of the world) (James, 2013; Bubic et al., 2010). In the context of RL, learning a dynamics model amounts to predicting the future state conditioned on the agent's action. This can be a very rich unsupervised learning signal by itself that encourages the agent to learn useful state representations. Besides, the agent can potentially use the learned model to improve exploration or perform planning by simulating the future. Although there has been a long history of work in this direction (Sutton, 1990; Sutton et al., 2008; Yao et al., 2009), most of the work considered relatively simple domains such as 2D grid-world where a linear function is expressive enough to represent a state-transition function. On the other hand, there has not been much work on how to build an accurate dynamics model of the environment and how to use it to improve control and planning on complex domains where the observations are high-dimensional.

In the first part of this thesis (Chapter III and Chapter IV), we present neural network architectures that learn to simulate the future in an action-conditional way and show how a learned model can be used to improve control. More specifically, we first discuss how to make reliable long-term predictions of high-dimensional observations using neural networks and show multiple ways to evaluate the quality and the usefulness of such a learned model on Atari games. In addition, this thesis develops a unified DRL framework that integrates both model-free and model-based RL approaches into a single neural network by jointly learning to predict the future and estimate the value of the state. More importantly, the proposed approach learns a dynamics model only in terms of rewards and values without needing to predict observations. We discuss how to perform lookahead planning with such a model and demonstrate the advantage of our approach compared to conventional model-based and model-free RL methods.

In addition to prediction and planning, the ability to quickly generalize to new situations or tasks based on prior experience is also one of the important problems in RL (Taylor and Stone, 2009; Lake et al., 2016). As a motivating example, humans can often easily find an exit in a new building, because we have learned a general strategy to navigate the 3D world and have knowledge about common building structures. We can also easily travel through a new city following a guidebook or use a new device (e.g., smartphone) based on prior knowledge or instructions (if available). Developing such a strong generalization ability is an important challenge for scaling up RL agents to a large number of states and tasks because the agent does not need any additional learning to handle new situations if it can generalize well. Although deep neural networks have been used to handle high-dimensional observations in RL, it remains an open question how to use deep learning to improve generalization ability in RL, which is the key motivation of using function approximation methods in RL (Sutton and Barto, 1998; Sutton, 1996).

The second part of the thesis (Chapter V and Chapter VI) develops several techniques to improve the generalization ability using deep neural networks. In particular, we consider partially observable environments where the agent should remember useful information from the past to solve a task. We introduce a set of cognitive tasks in a 3D environment and evaluates the agent's generalization performance on unseen 3D environments. We also present new memory-based architectures and demonstrate that the proposed architectures generalize better to unseen 3D worlds compared to existing baselines. In addition, this thesis considers a new generalization problem where the agent should learn to execute a set of tasks described by a form of instructions during training and generalize to unseen instructions during evaluation. To solve this problem, we propose a hierarchical architecture that learns to generalize by learning the relationship between different task descriptions. We demonstrate that the proposed architecture can generalize from a small set of tasks to a much larger set of tasks on a challenging 3D domain.

Finally, the trade-off between exploration and exploitation is another fundamental challenge in RL. In complex environments, where it is infeasible for the agent to explore the entire state-action space, it is important for the agent to efficiently explore the environment in order to discover the source of reward more often and more quickly. However, even with such an advanced exploration strategy, the frequency of receiving rewards can be still low if the reward signal is extremely delayed or sparse. In such cases, it may still take a huge amount of time for the agent to collect rewarding experiences and learn a good policy from them. Much of the prior work on exploration in RL has focused on different ways to provide exploration bonus reward to drive exploratory behavior based on curiosity or intrinsic motivation (Schmidhuber, 1991; Strehl and Littman, 2008; Bellemare et al., 2016). On the other hand, there has been relatively less understanding on how 'exploiting' good experiences can boost learning progress and how it affects future exploration.

The third part of the thesis (Chapter VII) studies how exploiting past good experiences can improve sample efficiency in complex environments. The main hypothesis is that exploiting good experiences (i.e., high-rewarding episodes) indirectly drives deep exploration and thus improves sample efficiency. To verify the hypothesis, we propose a new off-policy actor-critic algorithm, called self-imitation learning, which learns to reproduce past good experiences. We show that self-imitation learning allows the agent to quickly learn a good policy from a few good episodes and increases the chance of getting the next source of reward through further exploration. As a result, we demonstrate that self-imitation learning significantly reduces the sample complexity on a variety of domains including hard exploration Atari games.

To summarize, this thesis studies 1) how to build deep neural networks for learning a dynamics model of the environment for look-ahead planning, 2) how to generalize from prior experience to unseen partially observable environments and new tasks, and 3) how to exploit the agent's past experiences to drive deep exploration in RL.

1.1 Outline and Summary of Contributions

Chapter II describes background on deep reinforcement learning. Chapter III, IV, V, VI, and VII are the main contributions of the thesis. The main ideas and results for each contribution are described below. Chapter VIII summarizes the thesis and discusses future work.

Action-Conditional Video Prediction with Neural Networks (Chapter III)
This chapter considers a high-dimensional video prediction problem conditioned on action sequences. Not only does this amount to learning a dynamics model of the environment in RL, but also predicting high-dimensional video is itself interesting and challenging problem in deep learning and generative modeling. This chapter presents novel deep architectures that integrate a control variable (i.e., action) into a video prediction model. We show several ways to evaluate the usefulness of the action-conditional video prediction model and demonstrate that our architecture can predict more than 100 steps of visually-realistic future

frames on Atari games. In addition, this chapter shows how to use such a learned model to improve exploration in RL.

Value Prediction and Planning with Neural Networks (Chapter IV)
This chapter aims to answer an open research question on whether it is possible to perform lookahead planning without explicitly simulating observations. This is based on the premise that what we truly need for lookahead planning is the expected reward and value. To answer this question, this chapter presents a novel DRL architecture that integrates both model-free and model-based RL into a single neural network. Specifically, the proposed architecture learns to predict future rewards and values without predicting observations, and a state-transition model is implicitly learned through reward and value prediction objectives. This chapter then empirically shows that the proposed architecture has several advantages over both model-free and model-based RL architectures on a stochastic 2D domain and Atari games where building an accurate observation-prediction model is hard.

Neural Memory Architecture for Partially Observable Environment (Chapter V)
The ability to handle partially observable environments is a key challenge in RL because the agent is required to remember important information from the history of observations to make an optimal decision. This chapter introduces a set of challenging cognitive tasks in a 3D partially observable environment using Minecraft. The tasks require the agent to navigate in a 3D world given first-person-view observations and perform non-trivial reasoning (e.g., comparing visual patterns) to receive a positive reward. More importantly, the agent should deal with unseen and larger 3D worlds during evaluation, which requires memorizing important information for a longer time. This chapter systematically evaluates different deep architectures and shows that our proposed memory-based architectures can generalize much better to unseen and larger 3D environments than existing architectures.

Neural Hierarchical Architecture for Zero-Shot Task Generalization (Chapter VI)
In order for RL agents to be useful in real-world scenarios, the agent should be able to understand and execute many different tasks. More importantly, it is desirable for the agent to handle unseen tasks in a zero-shot way (e.g., a household robot that executes a variety of human user's natural language instructions). This chapter considers a new multi-task RL problem where the agent should execute different tasks depending on given task descriptions (i.e., instructions) and generalize to unseen and longer instructions during evaluation. To solve the problem, we present a hierarchical DRL architecture where a meta-controller passes a subtask to a low-level controller which executes it. Since it is infeasible to train

the agent on all possible combinations of instructions, we propose a new objective that allows the agent to learn all possible instructions without needing to experience them during training using metric learning techniques in deep learning.

Self-Imitation Learning (Chapter VII)
In complex environments where it is infeasible for the agent to explore the entire state-action space, achieving the balance between exploration and exploitation is crucial for learning a good policy within a reasonable amount of time. This chapter studies how exploiting past good experiences affects exploration and reduces sample complexity. More specifically, we propose self-imitation learning which exploits past good experiences by learning to reproduce them and demonstrate that self-imitation learning indirectly drives deep exploration and thus significantly improve sample efficiency on a variety of challenging RL domains such as hard exploration Atari games.

1.2 First Published Appearances of Contributions

Most of the contributions described in this thesis have been published at various venues. The following list describes the publications corresponding to each chapter:

- Chapter III: Oh, J., Guo, X., Lee, H., Lewis, R. L., and Singh, S. (2015). Action-conditional video prediction using deep networks in atari games. In *Advances in the Neural Information Processing System*.
- Chapter IV: Oh, J., Singh, S., and Lee, H. (2017a). Value prediction network. In *Advances in the Neural Information Processing System*.
- Chapter V: Oh, J., Chockalingam, V., Singh, S., and Lee, H. (2016). Control of memory, active perception, and action in minecraft. In *Proceedings of the International Conference on Machine Learning*.
- Chapter VI: Oh, J., Singh, S., Lee, H., and Kohli, P. (2017b). Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.
- Chapter VII: Oh, J., Guo, Y., Singh, S., and Lee, H. (2018). Self-imitation learning. In *Proceedings of the International Conference on Machine Learning*.

CHAPTER II

Background

2.1 Markov Decision Process

A Markov Decision Process (MDP) (Puterman, 2014) describes the interaction between an agent and a stochastic environment. Throughout this thesis, we consider a finite MDP which consists of:

- $\mathcal{S}$: A finite set of states of the environment.
- $\mathcal{A}$: A finite set of actions which the agent chooses.
- $P(r, s'|s, a) : \mathcal{S} \times \mathcal{A} \times \mathbb{R} \times \mathcal{S} \to [0, 1]$: A transition probability that the environment gives reward r and state s' for state s and action a.
- $\gamma \in [0, 1]$: A discount factor that defines the present value of the future rewards.

In a finite MDP,the agent observes its state $s_t \in \mathcal{S}$ at each time-step t, chooses an action $a_t \in \mathcal{A}$, receives a reward $r \in \mathbb{R}$, and observes the next state $s_{t+1} \in \mathcal{S}$. Such a sequential interaction between the agent and the environment results in a *trajectory* as follows:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, ...). \tag{2.1}$$

2.1.1 Policy and Value Functions

A stochastic policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a probability distribution over actions given a state, which is often represented as a conditional distribution $\pi(a|s)$. For a fixed policy π, a *value function* $V^\pi : \mathcal{S} \to \mathbb{R}$ and an *action-value function* (or Q-value function) $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

are defined as:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t \middle| \pi, s_0 = s\right] \tag{2.2}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t \middle| \pi, s_0 = s, a_0 = a\right]. \tag{2.3}$$

The goal of reinforcement learning (RL) is to find a policy π which maximizes the discounted sum of rewards (or value) as follows:

$$\operatorname*{argmax}_{\pi} V^{\pi}(s_0) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t \middle| \pi, s_0\right]. \tag{2.4}$$

2.1.2 Optimal Policies and Optimal Value Functions

There exists an *optimal policy* π which maximizes both $V^{\pi}(s)$ and $Q^{\pi}(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ (Puterman, 2014). The *optimal value function* $V^*(s)$ and the *optimal action-value function* $Q^*(s,a)$ are defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s), \forall s \in \mathcal{S} \tag{2.5}$$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{2.6}$$

$V^*(s)$ and $Q^*(s,a)$ satisfy *Bellman optimality equations* as follows:

$$V^*(s) = \max_{a \in \mathcal{A}} Q^*(s,a) \tag{2.7}$$

$$Q^*(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}\left[V^*(s')\right] \tag{2.8}$$

One can easily induce a deterministic optimal policy π^* given the optimal action-value function as follows:

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q^*(s,a) \tag{2.9}$$

2.2 Q-Learning

Q-learning (Watkins and Dayan, 1992) is an off-policy temporal-difference (TD) learning algorithm, which is designed to learn the optimal action-value function from trajectories.

2.2.1 Tabular Q-Learning

Given state transitions $s, a \rightarrow r, s'$ from *any* behavior policy, the tabular Q-learning algorithm updates an action-value function $Q(s, a)$ as follows:

$$Q(s,a) \leftarrow Q(s,a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right] \tag{2.10}$$

where η is a learning rate. It is shown that this update rule converges to the optimal action-value function with a minimal requirement that all state-action pairs continue to be updated.

2.2.2 Q-Learning with Function Approximation

Q-learning can be used with *function approximation* where the action-value function $Q_\theta(s, a)$ is represented by a function approximator parameterized by θ. More specifically, given state transitions $s, a \rightarrow r, s'$ from any behavior policy, the objective function of Q-learning with function approximation is:

$$\mathcal{L}^Q = \mathbb{E}_{s,a,r,s'} \left[\frac{1}{2} \| y - Q_\theta(s,a) \|^2 \right] \tag{2.11}$$

$$y = r + \gamma \max_{a'} Q_\theta(s', a') \tag{2.12}$$

$$\nabla_\theta \mathcal{L}^Q = \mathbb{E}_{s,a,r,s'} \left[(y - Q_\theta(s,a)) \nabla_\theta Q_\theta(s,a) \right], \tag{2.13}$$

where y is called *target Q-value*. Intuitively, we upate the parameter by taking a gradient descent using $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}^Q$ with a learning rate of η so that $Q_\theta(s, a)$ approximates the target Q-value. Unlike tabular Q-learning, however, Q-learning with function approximation does not guarantee convergence.

2.3 Policy Gradient

Policy gradient algorithms (Sutton et al., 1999a) directly compute the gradient of the expected sum of rewards with respect to the policy parameter θ using the *score function gradient estimator*. More formally, let $\pi_\theta(a|s)$ be a policy parameterized by θ. The gradient of the value of the policy $\pi_\theta(a|s)$ is given by:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\sum_a \nabla_\theta \pi_\theta(a|s_t) Q^{\pi_\theta}(s_t,a)\right] && (2.14)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\sum_a \frac{\nabla_\theta \pi_\theta(a|s_t)}{\pi_\theta(a|s_t)} Q^{\pi_\theta}(s_t,a)\pi_\theta(a|s_t)\right] && (2.15)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} Q^{\pi_\theta}(s_t,a_t)\right] && (2.16)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t,a_t)\right] && (2.17)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi(a_t|s_t) R_t\right], && (2.18)
\end{aligned}$$

where $R_t = \sum_{k=t}^{\infty}\gamma^{k-t} r_k$ is the return from a sample trajectory $\tau \sim \pi(a|s)$. We call this particular form of policy gradient REINFORCE (Williams, 1992). Intuitively, REINFORCE increases the probability of action a_t proportional to the return R_t.

2.3.1 Variance Reduction with Baseline

We can reduce the variance of the policy gradient using a state-dependent *baseline* $b(s_t)$:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right] &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\sum_a \nabla_\theta \pi_\theta(a|s_t) (Q^{\pi_\theta}(s_t,a) - b(s_t))\right] && (2.19)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty}\sum_a \frac{\nabla_\theta \pi_\theta(a|s_t)}{\pi_\theta(a|s_t)} (Q^{\pi_\theta}(s_t,a) - b(s_t))\pi_\theta(a|s_t)\right] && (2.20)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} (Q^{\pi_\theta}(s_t,a_t) - b(s_t))\right] && (2.21)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) (Q^{\pi_\theta}(s_t,a_t) - b(s_t))\right] && (2.22)\\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi(a_t|s_t) (R_t - b(s_t))\right]. && (2.23)
\end{aligned}$$

The baseline $b(s_t)$ can be any function, as long as it does not depend on actions. This is because of the following property:

$$\sum_a \nabla_\theta \pi_\theta(a|s_t) b(s_t) = b(s_t) \nabla_\theta \sum_a \pi_\theta(a|s_t) = b(s_t) \nabla_\theta 1 = 0. \tag{2.24}$$

A natural choice for the baseline is value-estimate $V_\theta(s_t) \approx V^{\pi_\theta}(s_t)$ which can be also learned by a function approximator θ. Since REINFORCE is an on-policy Monte-Carlo method for learning policy parameters, it is also natural to use the same samples for learning a policy and a value function for the baseline. A typical form of policy gradient with state-dependent baseline can be written as ($\sum_{t=0}^{\infty}$ is subsumed by $\mathbb{E}_{\pi_\theta}[\cdot]$ for brevity):

$$\mathcal{L}^{pg} = \mathbb{E}_{\pi_\theta}\left[\mathcal{L}^{pg}_{policy} + \beta \mathcal{L}^{pg}_{value}\right] \tag{2.25}$$

$$\mathcal{L}^{pg}_{policy} = -\nabla_\theta \log \pi(a_t|s_t)(R_t - V_\theta(s_t)) \tag{2.26}$$

$$\mathcal{L}^{pg}_{value} = \frac{1}{2}\|V_\theta(s_t) - R_t\|^2, \tag{2.27}$$

where β is the relative weight between the policy gradient objective and the value function objective. Intuitively, Equation 2.26 increases the probability of action a_t if the return is higher than expected ($R_t > V_\theta(s_t)$). Otherwise ($R_t < V_\theta(s_t)$), it decreases the probability.

2.3.2 Actor-Critic

Although the baseline technique reduces the variance of the policy gradient estimator, the term $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$ can have a high variance due to the stochasticity of the policy and the environment. The *actor-critic* algorithm further reduces the variance of the gradient through bootstrapping. More specifically, n-step actor-critic uses the following objective:

$$\mathcal{L}^{ac} = \mathbb{E}_{\pi_\theta}\left[\mathcal{L}^{ac}_{policy} + \beta \mathcal{L}^{pg}_{value}\right] \tag{2.28}$$

$$\mathcal{L}^{ac}_{policy} = -\nabla_\theta \log \pi(a_t|s_t)(R^n_t - V_\theta(s_t)) \tag{2.29}$$

$$\mathcal{L}^{ac}_{value} = \frac{1}{2}\|V_\theta(s_t) - R^n_t\|^2 \tag{2.30}$$

$$R^n_t = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\theta(s_{t+n}). \tag{2.31}$$

The only difference from Equation 2.25-2.27 is that we bootstrap the value at time-step $t+n$ instead of using the full return R_t. Though actor-critic introduces a bias proprotional to the value estimation error, it reduces the variance of the gradient, which turns out to be more sample-efficient in practice.

2.4 Deep Q-Network

2.4.1 Overview

Deep Q-network (DQN) (Mnih et al., 2015) is the first deep reinforcement learning architecture which uses a deep neural network as a value function approximator trained through Q-learning (Watkins and Dayan, 1992). In general, online Q-learning can be very unstable with non-linear function approximation (e.g., deep neural network). Deep Q-learning alleviates the instability issue using the following ideas.

- **Replay buffer**: DQN stores all transitions in a *replay buffer* (Lin, 1992) and performs Q-learning by randomly sampling a mini-batch of transitions from the replay buffer. Compared to on-policy transitions that are temporally correlated, random samples from the replay buffer are much less correlated.

- **Target network**: DQN uses a network with slightly outdated parameters called target network for computing target Q-value. This delays the effect of parameter updates and thus prevents rapid increment of Q-value estimates.

Algorithm 1 Deep Q-Learning

```
1:  Initialize parameter $\theta$
2:  Initialize target network parameter $\theta^-$
3:  Initialize replay buffer $\mathcal{B} \leftarrow \emptyset$
4:  for each iteration do
5:      # Collect samples
6:      $s \leftarrow$ Observe the current state
7:      $a \leftarrow$ Choose an action according to $\epsilon$-greedy policy
8:      $s', r \leftarrow$ Execute $a$ in the environment
9:      $\mathcal{B} \leftarrow \mathcal{B} \cup (s, a, r, s')$ Store transitions in the replay buffer
10:     # Update parameters
11:     Sample a mini-batch $B = \{(s, a, r, s')\}$ from the replay buffer $\mathcal{B}$
12:     Update the parameter $\theta$ using $\nabla_\theta \mathcal{L}^Q$ and $B$                (Eq 2.34)
13:     # Update target network parameters
14:     $\theta^- \leftarrow \theta$ after every $N_T$ steps
15: end for
```

2.4.2 Algorithm

Let $Q_\theta(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ be an action-value function represented by a neural network with a parameter θ. Deep Q-learning algorithm aims to learn the optimal action-value function by iteratively generating samples and updating the parameter θ as described in Algorithm 1 and further described below.

Generating experiences with ϵ-greedy policy Given a state $s \in \mathcal{S}$, the agent chooses an action by following ϵ-greedy policy which samples a random action $a \in \mathcal{A}$ with probability of ϵ or a greedy action $a = \operatorname{argmax}_{a'} Q_\theta(s, a')$ with probability of $1 - \epsilon$. Every state transition (s, a, r, s') is stored in the replay buffer ($\mathcal{B} = \{(s, a, r, s')\}$), which is used for learning.

Learning DQN updates the parameters using a mini-batch of transitions randomly sampled from the replay buffer $\mathcal{B}$ and using Q-learning objective $\mathcal{L}^Q$ as follows:

$$\mathcal{L}^Q = \mathbb{E}_{s,a,r,s' \sim \mathcal{B}} \left[\left(y - Q_\theta\left(s, a\right) \right)^2 \right] \tag{2.32}$$

$$\text{where } y = r + \gamma \max_{a'} Q_{\theta^-}(s', a') \tag{2.33}$$

$$\nabla_\theta \mathcal{L}^Q = \mathbb{E}_{s,a,r,s' \sim \mathcal{B}} \left[\left(y - Q_\theta\left(s, a\right) \right) \nabla_\theta Q_\theta(s, a) \right], \tag{2.34}$$

where $\gamma \in \mathbb{R}$ is a discount factor, and θ' is the parameter of the target network. The parameter of the target network (θ^-) is updated to the parameter (θ) after every N_T iterations.

2.4.3 Advanced DQNs

Double DQN van Hasselt (2010) observed that Q-learning tends to overestimate values and becomes over-optimistic. To remedy this, they proposed to decouple the selection of action from the evaluation when computing target Q-value. *Double DQN* (van Hasselt et al., 2016) implemented this idea by computing target Q-value as follows:

$$y = r + \gamma Q_{\theta^-}(s', \operatorname*{argmax}_{a'} Q_\theta(s', a')). \tag{2.35}$$

The only difference from DQN is that the target network (θ^-) is used to only evaluate the value of the next state, and the best action is selected according to the parameter θ.

Dueling Architecture Wang et al. (2016) proposed a new network architecture for DQN. Instead of directly producing $Q_\theta(s, a)$ as an output, *Dueling network* produces a value $V_\theta(s)$ and an advantage $A_\theta(s, a)$ as separate outputs. According to the definition of advantage, Q-value can be easily constructed as follows: $Q_\theta(s, a) = V_\theta(s) + A_\theta(s, a)$. Such a decomposition without any modification to learning algorithm turns out to make optimization easier and thus improve the performance of DQN.

Prioritized Experience Replay Schaul et al. (2016) implemented the idea of prioritized sweeping (Moore and Atkeson, 1993) in DQN. Instead of sampling uniformly from the replay buffer for learning, *prioritized replay* prioritizes samples according to the temporal-difference error (TD-error) $|y - Q_\theta(s, a)|$ in Equation 2.32 with the assumption that samples with high TD-errors are more informative for learning. It has been shown that prioritized sampling significantly improves DQN.

2.5 Parallel Methods for Advantage Actor-Critic

2.5.1 n-step Advantage Actor-Critic

Advantage actor-critic is a variant of policy-gradient method which learns both a policy and a value function as discussed in Chapter 2.5. More specifically, let $\pi_\theta(a|s) : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ and $V_\theta(s) : \mathcal{S} \rightarrow \mathbb{R}$ be a policy a value function parameterized by θ. Given a state s_t, the agent generates n-step trajectories by sampling actions from its policy $a \sim \pi_\theta(a|s)$. Given the n-step trajectory $\tau = (s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1}, ..., s_{t+n})$, n-step advantage actor-critic updates the parameter θ using the following objective:

$$\mathcal{L}^{ac} = \mathbb{E}_{\tau \sim \pi_\theta}\left[\mathcal{L}^{ac}_{policy} + \beta \mathcal{L}^{ac}_{value}\right] \tag{2.36}$$

$$\mathcal{L}^{ac}_{policy} = -\log \pi(a_t|s_t)(R^n - V_\theta(s_t)) - \alpha \mathcal{H}(\pi_\theta(a_t|s_t)) \tag{2.37}$$

$$\mathcal{L}^{ac}_{value} = \|V_\theta(s_t) - R^n\|^2, \tag{2.38}$$

where $R^n = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_\theta(s_{t+n})$ is a n-step bootstrapped return. $\mathcal{H}(\pi(a_t|s_t)) = -\log \pi(a_t|s_t)$ is the entropy of the policy, which encourages the policy to be uniform and prevents early convergence to a sub-optimal policy. The computation of actor-critic is easily parallelizable because it is on-policy algorithm which does not re-use past trajectories.

2.5.2 Parallel and Synchronous Method (A2C)

A parallel and synchronous implementation of advantage actor-critic algorithm (A2C) (Dhariwal et al., 2017) is described in Algorithm 2. The key idea is to use K parallel environments to execute actions in parallel. This makes the interaction between the agent and the environment very efficient because the agent can interact with many different environments in parallel (Line 7-9 in Algorithm 2). The rest of the algorithm including sampling actions and updating parameters is all synchronous. This means that the neural network operations (forward/backward pass) are executed in a single process which can effectively utilize

Algorithm 2 Synchronous Advantage Actor-Critic (A2C)

```
1: Initialize parameter $\theta$
2: Initialize $K$ parallel environments
3: for each iteration do
4:     for $n$ steps do
5:         Collect states $\mathbf{s} = \{s_k\}$ from each environment $k$
6:         Sample actions $\mathbf{a} \sim \pi_\theta(a|\mathbf{s})$
7:         for each parallel environment $k$ do                    (Parallel loop)
8:             Execute action $a_k$ in the environment
9:         end for
10:    end for
11:    Collect trajectories $\tau = \{\tau_k\}$ from each environment $k$
12:    Update the parameter $\theta$ using $\nabla_\theta \mathcal{L}^{ac}$ and trajectories $\tau$      (Equation 2.36)
13: end for
```

Algorithm 3 Asynchronous Advantage Actor-Critic (A3C)

```
1: Initialize parameter $\theta$
2: for each parallel thread $k$ do                                 (Parallel loop)
3:     Initialize thread-specific parameter $\theta_k$
4:     Initialize thread-specific environment
5:     for each iteration do
6:         Synchronize parameter $\theta_k \leftarrow \theta$
7:         Sample $n$-step trajectory $\tau \sim \pi_\theta(a|s)$
8:         Update the parameter $\theta$ using $\nabla_{\theta_k} \mathcal{L}^{ac}$ and trajectory $\tau$      (Equation 2.36)
9:     end for
10: end for
```

graphics processing unit (GPU).

2.5.3 Parallel and Asynchronous Method (A3C)

A parallel and asynchronous implementation of advantage actor-critic algorithm (A3C) (Mnih et al., 2016) is described in Algorithm 3. Unlike A2C, each thread in A3C has its own parameter θ_k which is synchronized with the global parameter θ after every iteration. Each thread compute the gradient of advantage actor-critic objective with its own trajectory and update the global parameter θ with its local gradient $\nabla_{\theta_k} \mathcal{L}^{ac}$. This is asynchronous because the thread-specific parameter θ_k can be slightly different from θ. This implementation better utilizes CPUs because each thread does not need to wait for the other threads except for parameter synchronization.

CHAPTER III

Action-Conditional Video Prediction with Neural Networks

Motivated by vision-based reinforcement learning (RL) problems, in particular Atari games from the recent benchmark Arcade Learning Environment (ALE), this chapter considers spatio-temporal prediction problems where future image-frames depend on control variables or actions as well as previous frames. While not composed of natural scenes, frames in Atari games are high-dimensional in size, can involve tens of objects with one or more objects being controlled by the actions directly and many other objects being influenced indirectly, can involve entry and departure of objects, and can involve deep partial observability. We propose and evaluate two deep neural network architectures that consist of encoding, action-conditional transformation, and decoding layers based on convolutional neural networks and recurrent neural networks. Experimental results show that the proposed architectures are able to generate visually-realistic frames that are also useful for control over approximately 100-step action-conditional futures in some games. To the best of our knowledge, this is the first work to make and evaluate long-term predictions on high-dimensional video conditioned by control inputs.

3.1 Introduction

Over the years, deep learning approaches (see Bengio (2009); Schmidhuber (2015) for survey) have shown great success in many visual perception problems (e.g., Krizhevsky et al. (2012); Ciresan et al. (2012); Szegedy et al. (2015); Girshick et al. (2014)). However, modeling videos (building a generative model) is still a very challenging problem because it often involves high-dimensional natural-scene data with complex temporal dynamics. Thus, recent studies have mostly focused on modeling simple video data, such as bouncing balls or small patches, where the next frame is highly-predictable given the previous frames (Sutskever et al., 2009; Mittelman et al., 2014; Michalski et al., 2014). In many applications, however, future frames depend not only on previous frames but also on control or action variables. For example, the first-person-view in a vehicle is affected by wheel-steering and acceleration. The camera observation of a robot is similarly dependent on its movement and changes of its camera angle. More generally, in vision-based reinforcement learning (RL) problems, learning to predict future images conditioned on actions amounts to learning a model of the dynamics of the agent-environment interaction, an essential component of model-based approaches to RL. In this chapter, we focus on Atari games from the Arcade Learning Environment (ALE) (Bellemare et al., 2013) as a source of challenging action-conditional video modeling problems. While not composed of natural scenes, frames in Atari games are high-dimensional, can involve tens of objects with one or more objects being controlled by the actions directly and many other objects being influenced indirectly, can involve entry and departure of objects, and can involve deep partial observability. To the best of our knowledge, this is the first work to make and evaluate long-term predictions on high-dimensional images conditioned by control inputs.

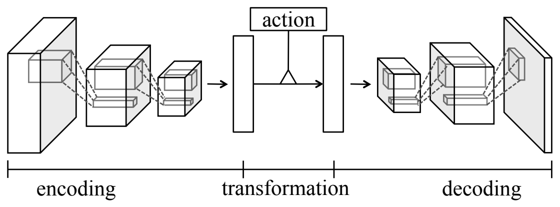
This chapter proposes, evaluates, and contrasts two spatio-temporal prediction architectures based on deep networks that incorporate action variables (See Figure 3.1). Our experimental results show that our architectures are able to generate realistic frames over 100-step action-conditional future frames without diverging in some Atari games. We show that the representations learned by our architectures 1) approximately capture natural similarity among actions, and 2) discover which objects are directly controlled by the agent's actions and which are only indirectly influenced or not controlled. We evaluated the usefulness of our architectures for control in two ways: 1) by replacing emulator frames with predicted frames in a previously-learned model-free controller (DQN; DeepMind's state of the art Deep-Q-Network for Atari Games (Mnih et al., 2013)), and 2) by using the predicted frames to drive a more informed than random exploration strategy to improve a model-free controller (also DQN).

3.2 Related Work

Video Prediction using Deep Networks. The problem of video prediction has led to a variety of architectures in deep learning. A recurrent temporal restricted Boltzmann machine (RTRBM) (Sutskever et al., 2009) was proposed to learn temporal correlations from sequential data by introducing recurrent connections in RBM. A structured RTRBM (sRTRBM) (Mittelman et al., 2014) scaled up RTRBM by learning dependency structures between observations and hidden variables from data. More recently, Michalski et al. (2014) proposed a higher-order gated autoencoder that defines multiplicative interactions between consecutive frames and mapping units, and showed that temporal prediction problem can be viewed as learning and inferring higher-order interactions between consecutive images. Srivastava et al. (2015) applied a sequence-to-sequence learning framework (Sutskever et al., 2014) to a video domain, and showed that long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) networks are capable of generating video of bouncing handwritten digits. In contrast to these previous studies, this chapter tackles problems where control variables affect temporal dynamics, and in addition scales up spatio-temporal prediction to larger-size images.

Combining Deep Learning and RL. Atari 2600 games provide challenging environments for RL because of high-dimensional visual observations, partial observability, and delayed rewards. Approaches that combine deep learning and RL have made significant advances (Mnih et al., 2013, 2015; Guo et al., 2014). Specifically, DQN (Mnih et al., 2013) combined Q-learning (Watkins and Dayan, 1992) with a convolutional neural network (CNN) and achieved state-of-the-art performance on many Atari games. Guo et al. (2014) used the ALE-emulator for making action-conditional predictions with slow UCT (Kocsis and Szepesvári, 2006), a Monte-Carlo tree search method, to generate training data for a fast-acting CNN, which outperformed DQN on several domains. Throughout this chapter we will use DQN to refer to the architecture used in Mnih et al. (2013) (a more recent work (Mnih et al., 2015) used a deeper CNN with more data to produce the currently best-performing Atari game players).

Action-Conditional Predictive Model for RL. The idea of building a predictive model for vision-based RL problems was introduced by Schmidhuber and Huber (1991). They proposed a neural network that predicts the attention region given the previous frame and an *attention-guiding* action. More recently, Lenz et al. (2015) proposed a recurrent neural network with multiplicative interactions that predicts the physical coordinate of a robot. Compared to this previous work, our work is evaluated on much higher-dimensional data with complex dependencies among observations. There have been a few attempts to learn from ALE data a transition-model that makes predictions of future frames. One line of work (Bellemare et al., 2013, 2014) divides game images into patches and applies a Bayesian framework to predict patch-based observations. However, this approach assumes that neighboring patches are enough to predict the center patch, which is not true in Atari games because of many complex interactions. The evaluation in this prior work is 1-step



(a) Feedforward encoding (b) Recurrent encoding

Figure 3.1: Proposed encoding-transformation-decoding network architectures.

prediction loss; in contrast, here we make and evaluate long-term predictions both for quality of pixels generated and for usefulness to control.

3.3 Proposed Architectures and Training Method

The goal of our architectures is to learn a function $f : \mathbf{x}_{1:t}, \mathbf{a}_t \rightarrow \mathbf{x}_{t+1}$, where $\mathbf{x}_t$ and $\mathbf{a}_t$ are the frame and action variables at time t, and $\mathbf{x}_{1:t}$ are the frames from time 1 to time t. Figure 3.1 shows our two architectures that are each composed of encoding layers that extract spatio-temporal features from the input frames (§3.3.1), action-conditional transformation layers that transform the encoded features into a prediction of the next frame in high-level feature space by introducing action variables as additional input (§3.3.2) and finally decoding layers that map the predicted high-level features into pixels (§3.3.3). Our contributions are in the novel action-conditional deep convolutional architectures for high-dimensional, long-term prediction as well as in the novel use of the architectures in vision-based RL domains.

3.3.1 Feedforward Encoding and Recurrent Encoding

Feedforward encoding takes a fixed history of previous frames as an input, which is concatenated through channels (Figure 3.1a), and stacked convolution layers extract spatio-temporal features directly from the concatenated frames. The encoded feature vector $\mathbf{h}_t^{enc} \in \mathbb{R}^n$ at time t is:

$$\mathbf{h}_t^{enc} = \text{CNN}\left(\mathbf{x}_{t-m+1:t}\right), \tag{3.1}$$

where $\mathbf{x}_{t-m+1:t} \in \mathbb{R}^{(m \times c) \times h \times w}$ denotes m frames of $h \times w$ pixel images with c color channels. CNN is a mapping from raw pixels to a high-level feature vector using multiple convolution layers and a fully-connected layer at the end, each of which is followed by a non-linearity. This encoding can be viewed as *early-fusion* (Karpathy et al., 2014) (other types of fusions, e.g., *late-fusion* or 3D convolution (Tran et al., 2015) can also be applied to this architecture).

Recurrent encoding takes one frame as an input for each time-step and extracts spatio-temporal features using an RNN in which the temporal dynamics is modeled by the recurrent layer on top of the high-level feature vector extracted by convolution layers (Figure 3.1b). In this chapter, LSTM without peephole connection is used for the recurrent layer as follows:

$$[\mathbf{h}_t^{enc}, \mathbf{c}_t] = \text{LSTM}\left(\text{CNN}\left(\mathbf{x}_t\right), \mathbf{h}_{t-1}^{enc}, \mathbf{c}_{t-1}\right), \tag{3.2}$$

where $\mathbf{c}_t \in \mathbb{R}^n$ is a *memory cell* that retains information from a deep history of inputs. Intuitively, CNN $(\mathbf{x}_t)$ is given as input to the LSTM so that the LSTM captures temporal correlations from high-level spatial features.

3.3.2 Multiplicative Action-Conditional Transformation

We use multiplicative interactions between the encoded feature vector and the control variables:

$$h_{t,i}^{dec} = \sum_{j,l} W_{ijl} h_{t,j}^{enc} a_{t,l} + b_i, \tag{3.3}$$

where $\mathbf{h}_t^{enc} \in \mathbb{R}^n$ is an encoded feature, $\mathbf{h}_t^{dec} \in \mathbb{R}^n$ is an action-transformed feature, $\mathbf{a}_t \in \mathbb{R}^a$ is the action-vector at time t, $\mathbf{W} \in \mathbb{R}^{n \times n \times a}$ is 3-way tensor weight, and $\mathbf{b} \in \mathbb{R}^n$ is bias. When the action **a** is represented using one-hot vector, using a 3-way tensor is equivalent to using different weight matrices for each action. This enables the architecture to model different transformations for different actions. The advantages of multiplicative interactions have been explored in image and text processing (Taylor and Hinton, 2009; Sutskever et al., 2011; Memisevic, 2013). In practice the 3-way tensor is not scalable because of its large number of parameters. Thus, we approximate the tensor by factorizing into three matrices as follows (Taylor and Hinton, 2009):

$$\mathbf{h}_t^{dec} = \mathbf{W}^{dec}\left(\mathbf{W}^{enc}\mathbf{h}_t^{enc} \odot \mathbf{W}^a \mathbf{a}_t\right) + \mathbf{b}, \tag{3.4}$$

where $\mathbf{W}^{dec} \in \mathbb{R}^{n \times f}$, $\mathbf{W}^{enc} \in \mathbb{R}^{f \times n}$, $\mathbf{W}^a \in \mathbb{R}^{f \times a}$, $\mathbf{b} \in \mathbb{R}^n$, and f is the number of factors. Unlike the 3-way tensor, the above factorization shares the weights between different actions by mapping them to the size-f factors. This sharing may be desirable relative to the 3-way tensor when there are common temporal dynamics in the data across different actions (discussed further in §3.4.3).

3.3.3 Convolutional Decoding

It has been recently shown that a CNN is capable of generating an image effectively using upsampling followed by convolution with stride of 1 (Dosovitskiy et al., 2015). Similarly, we use the “inverse" operation of convolution, called deconvolution, which maps 1×1 spatial region of the input to $d \times d$ using deconvolution kernels. The effect of $s \times s$ upsampling can be achieved without explicitly upsampling the feature map by using stride of s. We found that this operation is more efficient than upsampling followed by convolution because of the smaller number of convolutions with larger stride.

In the proposed architecture, the transformed feature vector $\mathbf{h}^{dec}$ is decoded into pixels as follows:

$$\hat{\mathbf{x}}_{t+1} = \text{Deconv}\left(\text{Reshape}\left(\mathbf{h}^{dec}\right)\right), \tag{3.5}$$

where Reshape is a fully-connected layer where hidden units form a 3D feature map, and Deconv consists of multiple deconvolution layers, each of which is followed by a non-linearity except for the last deconvolution layer.

3.3.4 Curriculum Learning with Multi-Step Prediction

It is almost inevitable for a predictive model to make noisy predictions of high-dimensional images. When the model is trained on a 1-step prediction objective, small prediction errors can compound through time. To alleviate this effect, we use a multi-step prediction objective. More specifically, given the training data $D = \left\{\left(\left(\mathbf{x}_1^{(i)}, \mathbf{a}_1^{(i)}\right), ..., \left(\mathbf{x}_{T_i}^{(i)}, \mathbf{a}_{T_i}^{(i)}\right)\right)\right\}_{i=1}^{N}$, the model is trained to minimize the average squared error over K-step predictions as follows:

$$\mathcal{L}_K(\theta) = \frac{1}{2K} \sum_i \sum_t \sum_{k=1}^{K} \left\| \hat{\mathbf{x}}_{t+k}^{(i)} - \mathbf{x}_{t+k}^{(i)} \right\|^2, \tag{3.6}$$

where $\hat{\mathbf{x}}_{t+k}^{(i)}$ is a k-step future prediction. Intuitively, the network is repeatedly *unrolled* through K time steps by using its prediction as an input for the next time-step.

The model is trained in multiple phases based on increasing K as suggested by Michalski et al. (2014). In other words, the model is trained to predict short-term future frames and fine-tuned to predict longer-term future frames after the previous phase converges. We found that this curriculum learning (Bengio et al., 2009) approach is necessary to stabilize the training. A stochastic gradient descent with backpropagation through time (BPTT) is used to optimize the parameters of the network.

3.4 Experiments

In the experiments that follow, we have the following goals for our two architectures. 1) To evaluate the predicted frames in two ways: qualitatively evaluating the generated video, and quantitatively evaluating the pixel-based squared error, 2) To evaluate the usefulness of predicted frames for control in two ways: by replacing the emulator's frames with predicted frames for use by DQN, and by using the predictions to improve exploration in DQN, and 3) To analyze the representations learned by our architectures. We begin by describing the details of the data, and model architecture, and baselines.

Data and Preprocessing. We used our replication of DQN to generate game-play video datasets using an ϵ-greedy policy with $\epsilon = 0.3$, i.e. DQN is forced to choose a random action with 30$\%$ probability. For each game, the dataset consists of about $500,000$ training frames and $50,000$ test frames with actions chosen by DQN. Following DQN, actions are chosen once every 4 frames which reduces the video from 60fps to 15fps. The number of actions available in games varies from 3 to 18, and they are represented as one-hot vectors. We used full-resolution RGB images (210×160) and preprocessed the images by subtracting mean pixel values and dividing each pixel value by 255.

Network Architecture. Across all game domains, we use the same network architecture as follows. The encoding layers consist of 4 convolution layers and one fully-connected layer with 2048 hidden units. The convolution layers use 64 (8×8), 128 (6×6), 128 (6×6), and 128 (4×4) filters with stride of 2. Every layer is followed by a rectified linear function (Nair and Hinton, 2010). In the recurrent encoding network, an LSTM layer with 2048 hidden units is added on top of the fully-connected layer. The number of factors in the transformation layer is 2048. The decoding layers consists of one fully-connected layer with 11264 $(= 128 \times 11 \times 8)$ hidden units followed by 4 deconvolution layers. The deconvolution layers use 128 (4×4), 128 (6×6), 128 (6×6), and 3 (8×8) filters with stride of 2. For the feedforward encoding network, the last 4 frames are given as an input for each time-step. The recurrent encoding network takes one frame for each time-step, but it is unrolled through the last 11 frames to initialize the LSTM hidden units before making a prediction. Our implementation is based on Caffe toolbox (Jia et al., 2014).

Details of Training. We use the curriculum learning scheme above with three phases of increasing prediction step objectives of 1, 3 and 5 steps, and learning rates of 10^{-4}, 10^{-5}, and 10^{-5}, respectively. RMSProp (Tieleman and Hinton, 2012; Graves, 2013) is used with momentum of 0.9, (squared) gradient momentum of 0.95, and min squared gradient of 0.01. The batch size for each training phase is 32, 8, and 8 for the feedforward encoding network and 4, 4, and 4 for the recurrent encoding network, respectively. When the recurrent encoding network is trained on 1-step prediction objective, the network is unrolled through 20 steps and predicts the last 10 frames by taking ground-truth images as input. Gradients are clipped at $[-0.1, 0.1]$ before non-linearity of each gate of LSTM as suggested by Graves (2013).

Two Baselines for Comparison. The first baseline is a multi-layer perceptron (*MLP*) that takes the last frame as input and has 4 hidden layers with 400, 2048, 2048, and 400 units. The action input is concatenated to the second hidden layer. This baseline uses approximately the same number of parameters as the recurrent encoding model. The second baseline, no-action feedforward (or *naFf*), is the same as the feedforward encoding model (Figure 3.1a) except that the transformation layer consists of one fully-connected layer that does not get the action as input.

3.4.1 Evaluation of Predicted Frames

Qualitative Evaluation: Prediction video. The prediction videos of our models and baselines are available at: `https://sites.google.com/a/umich.edu/junhyuk-oh/action-conditional-video-prediction`. As seen in the videos, the proposed models make qualitatively reasonable predictions over 30–500 steps depending on the game. In all games, the MLP baseline quickly diverges, and the naFf baseline fails to predict the controlled object. An example of long-term predictions is illustrated in Figure 3.2. We observed that both of our models predict complex local translations well such as the movement of vehicles and the controlled object. They can predict interactions between objects such as collision of two objects. Since our architectures effectively extract hierarchical features using CNN, they are able to make a prediction that requires a global context. For example, in Figure 3.2, the model predicts the sudden change of the location of the controlled object (from the top to the bottom) at 257-step.

However, both of our models have difficulty in accurately predicting small objects, such as bullets in Space Invaders. The reason is that the squared error signal is small when the model fails to predict small objects during training. Another difficulty is in handling stochasticity. In Seaquest, e.g., new objects appear from the left side or right side randomly, and so are hard to predict. Although our models do generate new objects with reasonable shapes and movements (e.g., after appearing they move as in the true frames), the generated

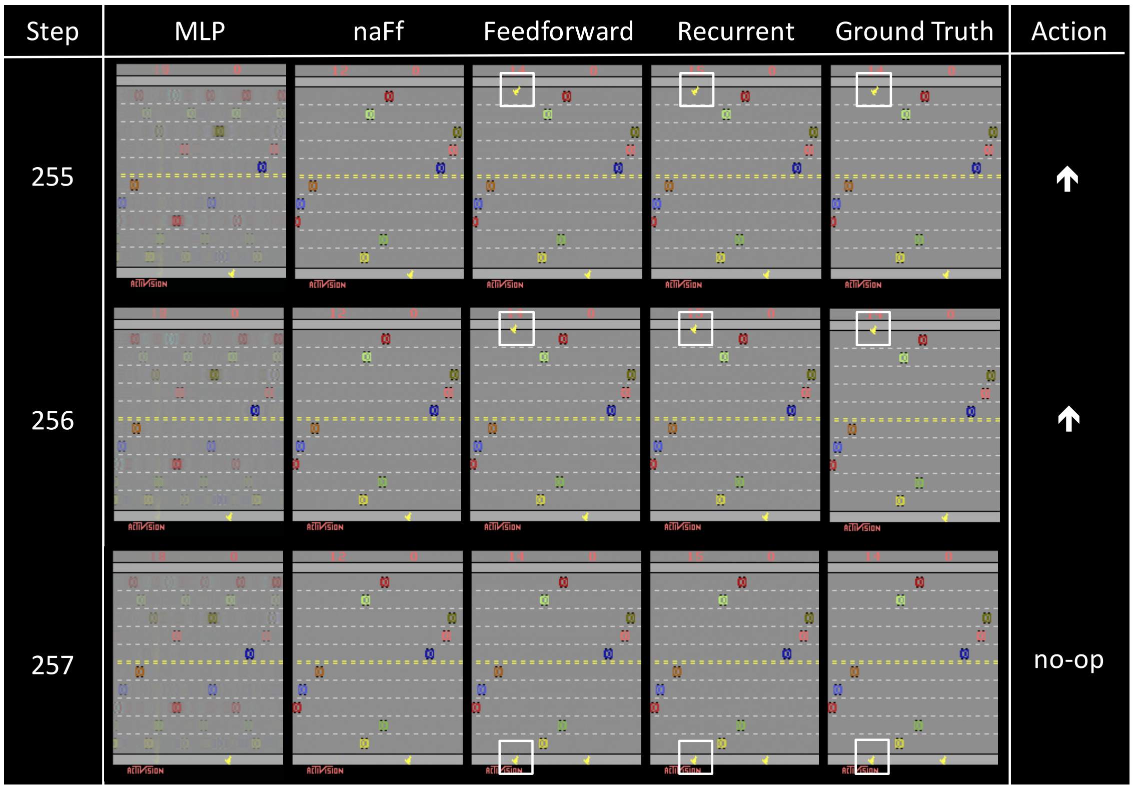


Figure 3.2: Example of predictions over 250 steps in Freeway. The 'Step' and 'Action' columns show the number of prediction steps and the actions taken respectively. The white boxes indicate the object controlled by the agent. From prediction step 256 to 257 the controlled object crosses the top boundary and reappears at the bottom; this non-linear shift is predicted by our architectures and is not predicted by MLP and naFf. The horizontal movements of the uncontrolled objects are predicted by our architectures and naFf but not by MLP.

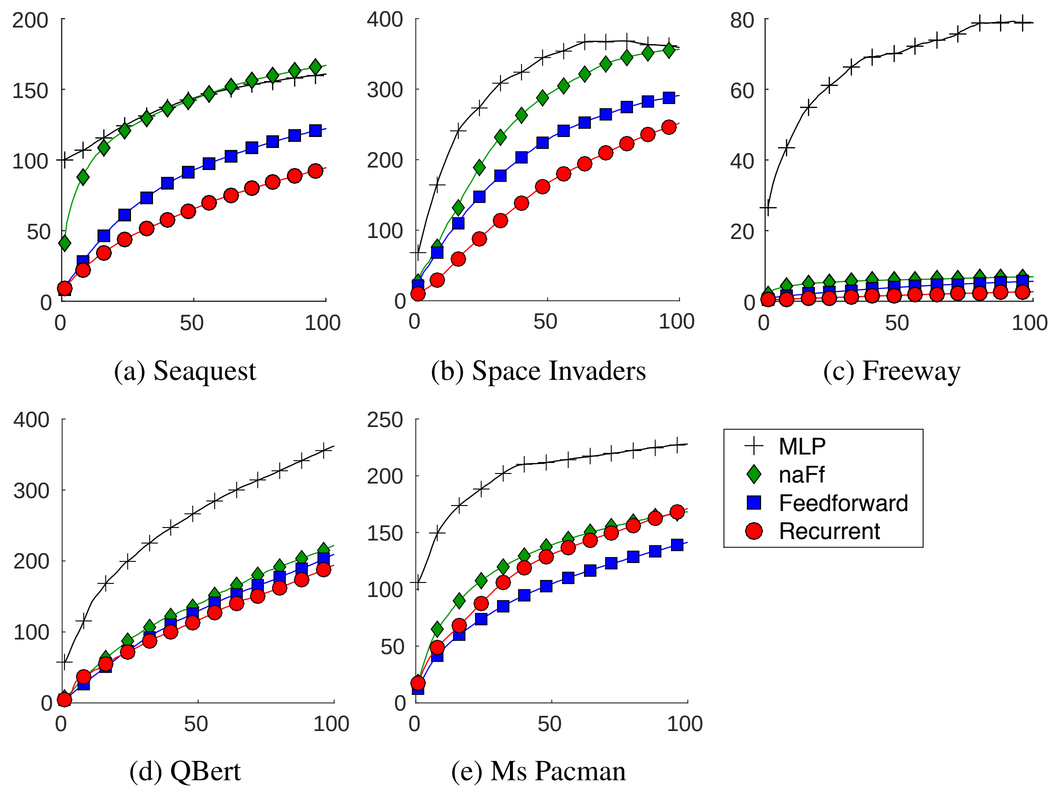
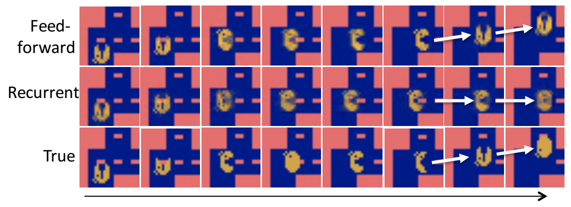


Figure 3.3: Mean squared error over 100-step predictions

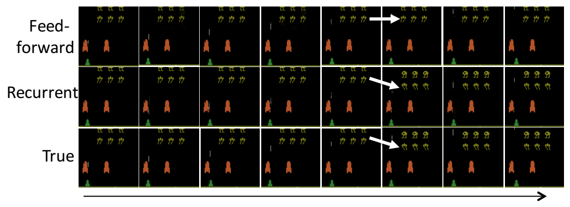
frames do not necessarily match the ground-truth.

Quantitative Evaluation: Squared Prediction Error. Mean squared error over 100-step predictions is reported in Figure 3.3. Our predictive models outperform the two baselines for all domains. However, the gap between our predictive models and naFf baseline is not large except for Seaquest. This is due to the fact that the object controlled by the action occupies only a small part of the image.

Qualitative Analysis of Relative Strengths and Weaknesses of Feedforward and Recurrent Encoding. We hypothesize that feedforward encoding can model more precise spatial transformations because its convolutional filters can learn temporal correlations directly from pixels in the concatenated frames. In contrast, convolutional filters in recurrent encoding can learn only spatial features from the one-frame input, and the temporal context has to be captured by the recurrent layer on top of the high-level CNN features without localized information. On the other hand, recurrent encoding is potentially better for modeling arbitrarily long-term dependencies, whereas feedforward encoding is not suitable for long-term dependencies because it requires more memory and parameters as more frames



(a) Ms Pacman (28 × 28 cropped)



(b) Space Invaders (90 × 90 cropped)

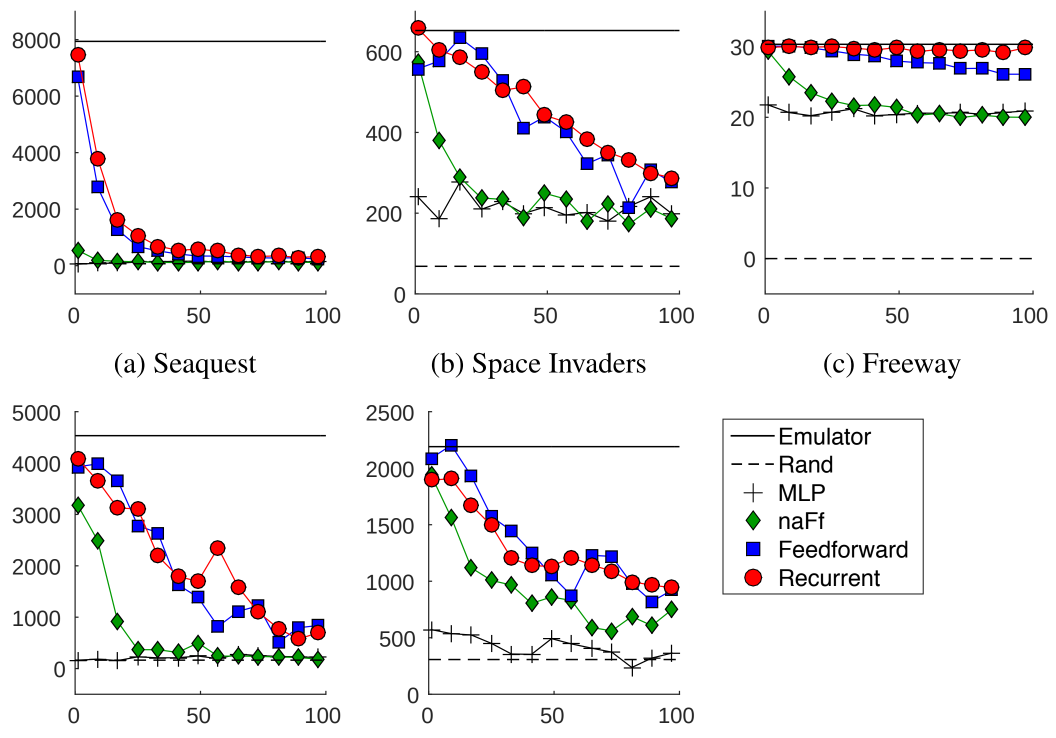
Figure 3.4: Comparison between two encoding models (feedforward and recurrent). (a) Controlled object is moving along a horizontal corridor. As the recurrent encoding model makes a small translation error at 4th frame, the true position of the object is in the crossroad while the predicted position is still in the corridor. The (true) object then moves upward which is not possible in the predicted position and so the predicted object keeps moving right. This is less likely to happen in feedforward encoding because its position prediction is more accurate. (b) The objects move down after staying at the same location for the first five steps. The feedforward encoding model fails to predict this movement because it only gets the last four frames as input, while the recurrent model predicts this downwards movement more correctly.

are concatenated into the input.

As evidence, in Figure 3.4a we show a case where feedforward encoding is better at predicting the precise movement of the controlled object, while recurrent encoding makes a 1-2 pixel translation error. This small error leads to entirely different predicted frames after a few steps. Since the feedforward and recurrent architectures are identical except for the encoding part, we conjecture that this result is due to the failure of precise spatio-temporal encoding in recurrent encoding. On the other hand, recurrent encoding is better at predicting when the enemies move in Space Invaders (Figure 3.4b). This is due to the fact that the enemies move after 9 steps, which is hard for feedforward encoding to predict because it takes only the last four frames as input. We observed similar results showing that feedforward encoding cannot handle long-term dependencies in other games.

3.4.2 Evaluating the Usefulness of Predictions for Control

Replacing Real Frames with Predicted Frames as Input to DQN. To evaluate how useful the predictions are for playing the games, we implement an evaluation method that uses the predictive model to replace the game emulator. More specifically, a DQN controller that takes the last four frames is first pre-trained using real frames and then used to play the games based on $\epsilon = 0.05$-greedy policy where the input frames are generated by our predictive model instead of the game emulator. To evaluate how the depth of predictions influence the quality of control, we re-initialize the predictions using the true last frames



(a) Seaquest (b) Space Invaders (c) Freeway

(d) QBert (e) Ms Pacman

Figure 3.5: Game play performance using the predictive model as an emulator. ‘Emulator’ and ‘Rand’ correspond to the performance of DQN with true frames and random play respectively. The x-axis is the number of steps of prediction before re-initialization. The y-axis is the average game score measured from 30 plays.

Table 3.1: Average game score of DQN over 100 plays with standard error. The first row and the second row show the performance of our DQN replication with different exploration strategies.

Model	Seaquest	S. Invaders	Freeway	QBert	Ms Pacman
Random exploration	13119 (538)	698 (20)	30.9 (0.2)	3876 (106)	2281 (53)
Informed exploration	13265 (577)	681 (23)	32.2 (0.2)	8238 (498)	2522 (57)

after every n-steps of prediction for $1 \leq n \leq 100$. Note that the DQN controller never takes a true frame, just the outputs of our predictive models.

The results are shown in Figure 3.5. Unsurprisingly, replacing real frames with predicted frames reduces the score. However, in all the games using the model to repeatedly predict only a few time steps yields a score very close to that of using real frames. Our two architectures produce much better scores than the two baselines for deep predictions than would be suggested based on the much smaller differences in squared error. The likely cause of this is that our models are better able to predict the movement of the controlled object relative to the baselines even though such an ability may not always lead to better squared error. In three out of the five games the score remains much better than the score of random play even when using 100 steps of prediction.

Improving DQN via Informed Exploration. To learn control in an RL domain, exploration of actions and states is necessary because without it the agent can get stuck in a bad sub-optimal policy. In DQN, the CNN-based agent was trained using an ϵ-greedy policy in which the agent chooses either a greedy action or a random action by flipping a coin with probability of ϵ. Such random exploration is a basic strategy that produces sufficient exploration, but can be slower than more informed exploration strategies. Thus, we propose an *informed exploration* strategy that follows the ϵ-greedy policy, but chooses exploratory actions that lead to a frame that has been visited least often (in the last d time steps), rather than random actions. Implementing this strategy requires a predictive model because the next frame for each possible action has to be considered.

The method works as follows. The most recent d frames are stored in a *trajectory memory*, denoted $D = \left\{\mathbf{x}^{(i)}\right\}_{i=1}^{d}$. The predictive model is used to get the next frame $\mathbf{x}^{(a)}$ for every action a. We estimate the visit-frequency for every predicted frame by summing the similarity between the predicted frame and the most d recent frames stored in the trajectory

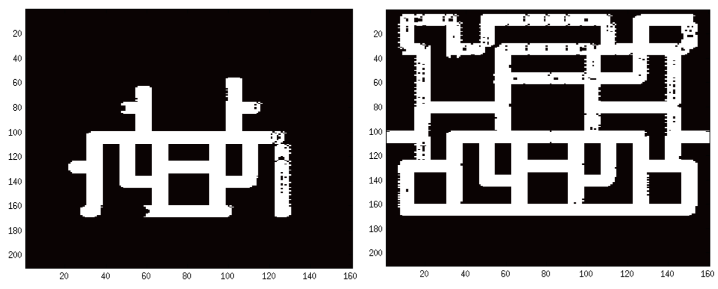

(a) Random exploration. (b) Informed exploration.

Figure 3.6: Comparison between two exploration methods on Ms Pacman. Each heat map shows the trajectories of the controlled object measured over 2500 steps for the corresponding method.

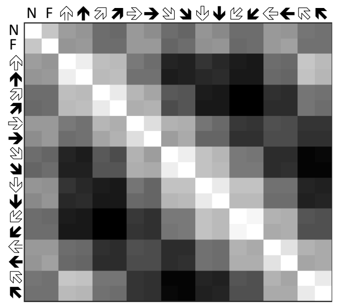


Figure 3.7: Cosine similarity between every pair of action factors (see text for details).

memory using a Gaussian kernel as follows:

$$n_D(\mathbf{x}^{(a)}) = \sum_{i=1}^{d} k(\mathbf{x}^{(a)}, \mathbf{x}^{(i)}) \tag{3.7}$$

$$k(\mathbf{x}, \mathbf{y}) = \exp(-\sum_{j} \min(\max((x_j - y_j)^2 - \delta, 0), 1)/\sigma) \tag{3.8}$$

where δ is a threshold, and σ is a kernel bandwidth. The trajectory memory size is 200 for QBert and 20 for the other games, $\delta = 0$ for Freeway and 50 for the others, and $\sigma = 100$ for all games. For computational efficiency, we trained a new feedforward encoding network on 84×84 gray-scaled images as they are used as input for DQN. Table 3.1 summarizes the results. The informed exploration improves DQN's performance using our predictive model in three of five games, with the most significant improvement in QBert. Figure 3.6 shows how the informed exploration strategy improves the initial experience of DQN.

3.4.3 Analysis of Learned Representations

Similarity among Action Representations. In the factored multiplicative interactions, every action is linearly transformed to f factors ($\mathbf{W}^a\mathbf{a}$ in Equation 3.4). In Figure 3.7 we present the cosine similarity between every pair of action-factors after training in Seaquest. 'N' and 'F' corresponds to 'no-operation' and 'fire'. Arrows correspond to movements with (black) or without (white) 'fire'. There are positive correlations between actions that have the same movement directions (e.g., 'up' and 'up+fire'), and negative correlations between actions that have opposing directions. These results are reasonable and discovered automatically in learning good predictions.

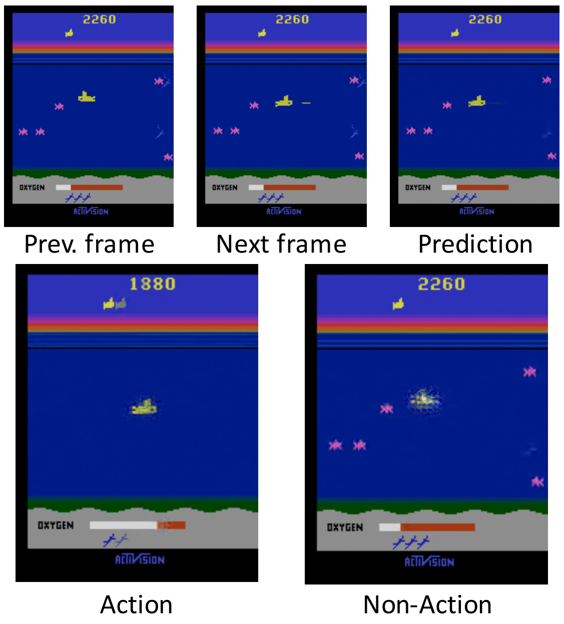


Figure 3.8: Distinguishing controlled and uncontrolled objects. *Action* image shows a prediction given only learned action-factors with high variance; *Non-Action* image given only low-variance factors.

Distinguishing Controlled and Uncontrolled Objects is itself a hard and interesting problem. Bellemare et al. (2012) proposed a framework to learn *contingent regions* of an image affected by agent action, suggesting that contingency awareness is useful for model-free agents. We show that our architectures implicitly learn contingent regions as they learn to predict the entire image.

In our architectures, a factor ($f_i = (\mathbf{W}^a_{i,:})^\top \mathbf{a}$) with higher variance measured over all possible actions, $\text{Var}\left(f_i\right) = \mathbb{E}_\mathbf{a}\left[(f_i - \mathbb{E}_\mathbf{a}[f_i])^2\right]$, is more likely to transform an image differently depending on actions, and so we assume such factors are responsible for transforming the parts of the image related to actions. We therefore collected the high variance (referred to as "highvar") factors from the model trained on Seaquest (around 40% of factors), and collected the remaining factors into a low variance ("lowvar") subset. Given an image and an action, we did two controlled forward propagations: giving only highvar factors (by setting the other factors to zeros) and vice versa. The results are visualized as 'Action' and 'Non-Action' in Figure 3.8. Interestingly, given only highvar-factors (Action), the model predicts sharply the movement of the object controlled by actions, while the other parts are mean pixel values. In contrast, given only lowvar-factors (Non-Action), the model predicts the movement of the other objects and the background (e.g., oxygen), and the controlled object stays at its previous location. This result implies that our model learns to distinguish between controlled objects and uncontrolled objects and transform them using disentangled representations (see Rifai et al. (2012); Reed et al. (2014); Yang et al. (2015) for related work on disentangling factors of variation).

3.5 Discussion

This chapter introduced two different novel deep architectures that predict future frames that are dependent on actions and showed qualitatively and quantitatively that they are able to predict visually-realistic and useful-for-control frames over 100-step futures on several Atari game domains. To our knowledge, this is the first to show good deep predictions in Atari games. Since our architectures were domain independent we expect that they will generalize to many vision-based RL problems. In future work we will learn models that predict future reward in addition to predicting future frames and evaluate the performance of our architectures in model-based RL.

CHAPTER IV

Value Prediction and Planning with Neural Networks

This chapter presents a novel deep reinforcement learning (RL) architecture, called Value Prediction Network (VPN), which integrates model-free and model-based RL methods into a single neural network. In contrast to typical model-based RL methods, VPN learns a dynamics model whose abstract states are trained to make option-conditional predictions of future values (discounted sum of rewards) rather than of future observations. Our experimental results show that VPN has several advantages over both model-free and model-based baselines in a stochastic environment where careful planning is required but building an accurate observation-prediction model is difficult. Furthermore, VPN outperforms Deep Q-Network (DQN) on several Atari games even with short-lookahead planning, demonstrating its potential as a new way of learning a good state representation.

4.1 Introduction

Model-based reinforcement learning (RL) approaches attempt to learn a model that predicts future observations conditioned on actions and can thus be used to simulate the real environment and do multi-step lookaheads for planning. We will call such models an *observation-prediction model* to distinguish it from another form of model introduced in this chapter. Building an accurate observation-prediction model is often very challenging when the observation space is large (Oh et al., 2015; Finn et al., 2016; Kalchbrenner et al., 2016; Chiappa et al., 2017) (e.g., high-dimensional pixel-level image frames), and even more difficult when the environment is stochastic. Therefore, a natural question is whether it is possible to plan without predicting future observations.

In fact, raw observations may contain information unnecessary for planning, such as dynamically changing backgrounds in visual observations that are irrelevant to their value/utility. The starting point of this work is the premise that what planning truly requires is the ability to predict the rewards and values of future states. An observation-prediction model relies on its predictions of observations to predict future rewards and values. What if we could predict future rewards and values directly without predicting future observations? Such a model could be more easily learnable for complex domains or more flexible for dealing with stochasticity. In this chapter, we address the problem of learning and planning from a *value-prediction model* that can directly generate/predict the value/reward of future states without generating future observations.

Our main contribution is a novel neural network architecture we call the *Value Prediction Network* (VPN). The VPN combines model-based RL (i.e., learning the dynamics of an abstract state space sufficient for computing future rewards and values) and model-free RL (i.e., mapping the learned abstract states to rewards and values) in a unified framework. In order to train a VPN, we propose a combination of *temporal-difference search* (Silver et al., 2012) (TD search) and n-step Q-learning (Mnih et al., 2016). In brief, VPNs learn to predict values via Q-learning and rewards via supervised learning. At the same time, VPNs perform lookahead planning to choose actions and compute bootstrapped target Q-values.

Our empirical results on a 2D navigation task demonstrate the advantage of VPN over model-free baselines (e.g., Deep Q-Network (Mnih et al., 2015)). We also show that VPN is more robust to stochasticity in the environment than an observation-prediction model approach. Furthermore, we show that our VPN outperforms DQN on several Atari games (Bellemare et al., 2013) even with short-lookahead planning, which suggests that our approach can be potentially useful for learning better abstract-state representations and reducing sample-complexity.

4.2 Related Work

Model-based Reinforcement Learning. Dyna-Q (Sutton, 1990; Sutton et al., 2008; Yao et al., 2009) integrates model-free and model-based RL by learning an observation-prediction model and using it to generate samples for Q-learning in addition to the model-free samples obtained by acting in the real environment. Gu et al. (Gu et al., 2016) extended these ideas to continuous control problems. Our work is similar to Dyna-Q in the sense that planning and learning are integrated into one architecture. However, VPNs perform a lookahead tree search to choose actions and compute bootstrapped targets, whereas Dyna-Q uses a learned model to generate imaginary samples. In addition, Dyna-Q learns a model of the environment separately from a value function approximator. In contrast, the dynamics model in VPN is combined with the value function approximator in a single neural network and indirectly learned from reward and value predictions through backpropagation.

Another line of work Oh et al. (2015); Chiappa et al. (2017); Guo et al. (2016); Stadie et al. (2015) uses observation-prediction models not for planning, but for improving exploration. A key distinction from these prior works is that our method learns abstract-state dynamics not to predict future observations, but instead to predict future rewards/values. For continuous control problems, deep learning has been combined with model predictive control (MPC) Finn and Levine (2017); Lenz et al. (2015); Raiko and Tornio (2009), a specific way of using an observation-prediction model. In cases where the observation-prediction model is differentiable with respect to continuous actions, backpropagation can be used to find the optimal action Mishra et al. (2017) or to compute value gradients Heess et al. (2015). In contrast, our work focuses on learning and planning using lookahead for discrete control problems.

Our VPNs are related to Value Iteration Networks Tamar et al. (2016) (VINs) which perform value iteration (VI) by approximating the Bellman-update through a convolutional neural network (CNN). However, VINs perform VI over the entire state space, which in practice requires that 1) the state space is small and representable as a vector with each dimension corresponding to a separate state and 2) the states have a topology with local transition dynamics (e.g., 2D grid). VPNs do not have these limitations and are thus more generally applicable, as we will show empirically in this chapter.

VPN is close to and in-part inspired by Predictron Silver et al. (2017b) in that a recurrent neural network (RNN) acts as a transition function over abstract states. VPN can be viewed as a *grounded* Predictron in that each rollout corresponds to the transition in the environment, whereas each rollout in Predictron is purely abstract. In addition, Predictrons are limited to uncontrolled settings and thus policy evaluation, whereas our VPNs can learn an optimal policy in controlled settings.

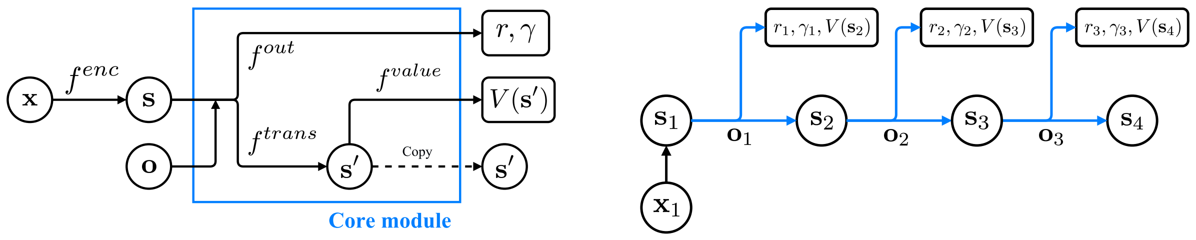
Model-free Deep Reinforcement Learning. Mnih et al. Mnih et al. (2015) proposed the Deep Q-Network (DQN) architecture which learns to estimate Q-values using deep neural networks. A lot of variations of DQN have been proposed for learning better state representation Wang et al. (2016); Kulkarni et al. (2016b); Hausknecht and Stone (2015); Oh et al. (2016); Vezhnevets et al. (2016); Parisotto and Salakhutdinov (2018), including the use of memory-based networks for handling partial observability Hausknecht and Stone (2015); Oh et al. (2016); Parisotto and Salakhutdinov (2018), estimating both state-values and advantage-values as a decomposition of Q-values Wang et al. (2016), learning successor state representations Kulkarni et al. (2016b), and learning several auxiliary predictions in addition to the main RL values Jaderberg et al. (2017). Our VPN can be viewed as a model-free architecture which 1) decomposes Q-value into reward, discount, and the value of the next state and 2) uses multi-step reward/value predictions as auxiliary tasks to learn a good representation. A key difference from the prior work listed above is that our VPN learns to simulate the future rewards/values which enables planning. Although STRAW Vezhnevets et al. (2016) can maintain a sequence of future actions using an external memory, it cannot explicitly perform planning by simulating future rewards/values.

Monte-Carlo Planning. Monte-Carlo Tree Search (MCTS) methods (Kocsis and Szepesvári, 2006; Browne et al., 2012) have been used for complex search problems, such as the game of Go, where a simulator of the environment is already available and thus does not have to be learned. Most recently, *AlphaGo* Silver et al. (2016) introduced a *value network* that directly estimates the value of state in Go in order to better approximate the value of leaf-node states during tree search. Our VPN takes a similar approach by predicting the value of abstract future states during tree search using a value function approximator. *Temporal-difference search* Silver et al. (2012) (TD search) combined TD-learning with MCTS by computing target values for a value function approximator through MCTS. Our algorithm for training VPN can be viewed as an instance of TD search, but it learns the dynamics of future rewards/values instead of being given a simulator.

4.3 Value Prediction Network

The value prediction network is developed for semi-Markov decision processes (SMDPs). Let $\mathbf{x}_t$ be the observation or a history of observations for partially observable MDPs (henceforth referred to as just observation) and let $\mathbf{o}_t$ be the *option* (Sutton et al., 1999b; Stolle and Precup, 2002; Precup, 2000) at time t. Each option maps observations to primitive actions, and the following Bellman equation holds for all policies π: $Q^{\pi}(\mathbf{x}_t, \mathbf{o}_t) = \mathbb{E}[\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V^{\pi}(\mathbf{x}_{t+k})]$, where γ is a discount factor, r_t is the immediate reward at time t, and k is the number of time steps taken by the option $\mathbf{o}_t$ before terminating in observation $\mathbf{x}_{t+k}$.

A VPN not only learns an option-value function $Q_{\theta}\left(\mathbf{x}_t, \mathbf{o}_t\right)$ through a neural network parameterized by θ like model-free RL, but also learns the dynamics of the rewards/values to perform planning. We describe the architecture of VPN in Section 4.3.1. In Section 4.3.2, we describe how to perform planning using VPN. Section 4.3.3 describes how to train VPN in a Q-Learning-like framework Watkins and Dayan (1992).



(a) One-step rollout (b) Multi-step rollout

Figure 4.1: Value prediction network. (a) VPN learns to predict immediate reward, discount, and the value of the next abstract-state. (b) VPN unrolls the core module in the abstract-state space to compute multi-step rollouts.

4.3.1 Architecture

The VPN consists of the following modules parameterized by $\theta = \{\theta^{enc}, \theta^{value}, \theta^{out}, \theta^{trans}\}$:

$$\textbf{Encoding } f_\theta^{enc} : \mathbf{x} \mapsto \mathbf{s} \qquad \textbf{Value } f_\theta^{value} : \mathbf{s} \mapsto V_\theta(\mathbf{s})$$
$$\textbf{Outcome } f_\theta^{out} : \mathbf{s}, \mathbf{o} \mapsto r, \gamma \qquad \textbf{Transition } f_\theta^{trans} : \mathbf{s}, \mathbf{o} \mapsto \mathbf{s}'$$

- **Encoding** module maps the observation ($\mathbf{x}$) to the abstract state ($\mathbf{s} \in \mathbb{R}^m$) using neural networks (e.g., CNN for visual observations). Thus, $\mathbf{s}$ is an *abstract-state representation which will be learned by the network (and not an environment state or even an approximation to one).*

- **Value** module estimates the value of the abstract-state ($V_\theta(\mathbf{s})$). Note that the value module is not a function of the observation, but a function of the abstract-state.

- **Outcome** module predicts the option-reward ($r \in \mathbb{R}$) for executing the option $\mathbf{o}$ at abstract-state $\mathbf{s}$. If the option takes k primitive actions before termination, the outcome module should predict the discounted sum of the k immediate rewards as a scalar. The outcome module also predicts the option-discount ($\gamma \in \mathbb{R}$) induced by the number of steps taken by the option.

- **Transition** module transforms the abstract-state to the next abstract-state ($\mathbf{s}' \in \mathbb{R}^m$) in an option-conditional manner.

Figure 4.1a illustrates the *core* module which performs 1-step rollout by composing the above modules: $f_\theta^{core} : \mathbf{s}, \mathbf{o} \mapsto r, \gamma, V_\theta(\mathbf{s}'), \mathbf{s}'$. The core module takes an abstract-state and option as input and makes separate option-conditional predictions of the option-reward (henceforth, reward), the option-discount (henceforth, discount), and the value of the abstract-state at option-termination. By combining the predictions, we can estimate the Q-value as follows: $Q_\theta(\mathbf{s}, \mathbf{o}) = r + \gamma V_\theta(\mathbf{s}')$. In addition, the VPN recursively applies the core module to predict the sequence of future abstract-states as well as rewards and discounts given an initial abstract-state and a sequence of options as illustrated in Figure 4.1b.

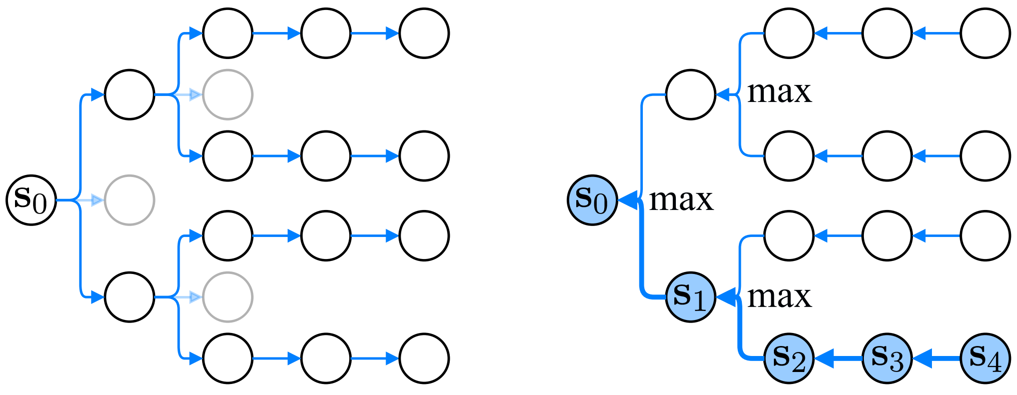
4.3.2 Planning

VPN has the ability to simulate the future and plan based on the simulated future abstract-states. Although many existing planning methods (e.g., MCTS) can be applied to the VPN, we implement a simple planning method which performs rollouts using the VPN up to a certain depth (say d), henceforth denoted as *planning depth*, and aggregates all intermediate value estimates as described in Algorithm 4 and Figure 4.2. More formally, given an abstract-state $\mathbf{s} = f_\theta^{enc}(\mathbf{x})$ and an option $\mathbf{o}$, the Q-value calculated from d-step planning is defined as:

$$Q_\theta^d(\mathbf{s}, \mathbf{o}) = r + \gamma V_\theta^d(\mathbf{s}') \qquad V_\theta^d(\mathbf{s}) = \begin{cases} V_\theta(\mathbf{s}) & \text{if } d = 1 \\ \frac{1}{d} V_\theta(\mathbf{s}) + \frac{d-1}{d} \max_{\mathbf{o}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{o}) & \text{if } d > 1, \end{cases} \tag{4.1}$$

where $\mathbf{s}' = f_\theta^{trans}(\mathbf{s}, \mathbf{o}), V_\theta(\mathbf{s}) = f_\theta^{value}(\mathbf{s})$, and $r, \gamma = f_\theta^{out}(\mathbf{s}, \mathbf{o})$. Our planning algorithm is divided into two steps: expansion and backup. At the expansion step (see Figure 4.2a), we recursively simulate options up to a depth of d by unrolling the core module. At the backup step, we compute the weighted average of the direct value estimate $V_\theta(\mathbf{s})$ and $\max_{\mathbf{o}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{o})$ to compute $V_\theta^d(\mathbf{s})$ (i.e., value from d-step planning) in Equation 4.1. Note that $\max_{\mathbf{o}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{o})$ is the average over $d-1$ possible value estimates. We propose to compute the uniform average over all possible returns by using weights proportional to 1 and $d-1$ for $V_\theta(\mathbf{s})$ and $\max_{\mathbf{o}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{o})$ respectively. Thus, $V_\theta^d(\mathbf{s})$ is the uniform average of d expected returns along the path of the best sequence of options as illustrated in Figure 4.2b.

To reduce the computational cost, we simulate only b-best options at each expansion step based on $Q^1(\mathbf{s}, \mathbf{o})$. We also find that choosing only the best option after a certain depth does not compromise the performance much, which is analogous to using a default policy in MCTS beyond a certain depth. This heuristic visits reasonably good abstract states during planning, though a more principled way such as UCT Kocsis and Szepesvári (2006) can also be used to balance exploration and exploitation. This planning method is used for choosing options and computing target Q-values during training, as described in the following section.



(a) Expansion (b) Backup

Figure 4.2: Planning with VPN. (a) Simulate b-best options up to a certain depth ($b = 2$ in this example). (b) Aggregate all possible returns along the best sequence of future options.

Algorithm 4 Q-value from d-step planning

```
function Q-PLAN($\mathbf{s}, \mathbf{o}, d$)
    $r, \gamma, V(\mathbf{s}'), \mathbf{s}' \leftarrow f_\theta^{core}(\mathbf{s}, \mathbf{o})$
    if $d = 1$ then
        return $r + \gamma V(\mathbf{s}')$
    end if
    $\mathcal{A} \leftarrow b$-best options based on $Q^1(\mathbf{s}', \mathbf{o}')$
    for $\mathbf{o}' \in \mathcal{A}$ do
        $q_{\mathbf{o}'} \leftarrow$ Q-PLAN($\mathbf{s}', \mathbf{o}', d-1$)
    end for
    return $r + \gamma \left[\frac{1}{d} V(\mathbf{s}') + \frac{d-1}{d} \max_{\mathbf{o}' \in \mathcal{A}} q_{\mathbf{o}'}\right]$
end function
```

4.3.3 Learning

VPN can be trained through any existing value-based RL algorithm for the value predictions combined with supervised learning for reward and discount predictions. In this chapter, we present a modification of n-step Q-learning (Mnih et al., 2016) and TD search (Silver et al., 2012). The main idea is to generate trajectories by following ϵ-greedy policy based on the planning method described in Section 4.3.2. Given an n-step trajectory $\mathbf{x}_1, \mathbf{o}_1, r_1, \gamma_1, \mathbf{x}_2, \mathbf{o}_2, r_2, \gamma_2, ..., \mathbf{x}_{n+1}$ generated by the ϵ-greedy policy, k-step predictions are

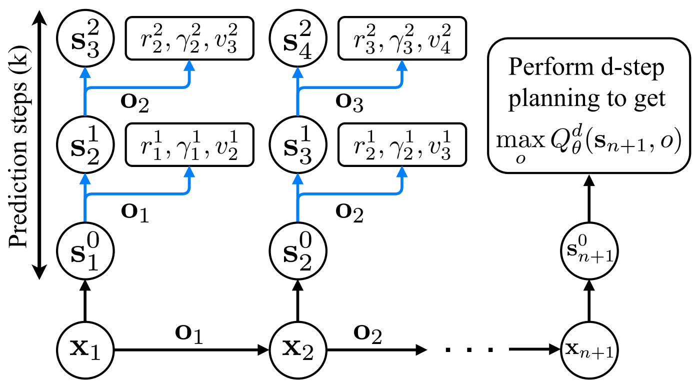


Figure 4.3: Illustration of learning process.

defined as follows:

$$\mathbf{s}_t^k = \begin{cases} f_\theta^{enc}(\mathbf{x}_t) & \text{if } k = 0 \\ f_\theta^{trans}(\mathbf{s}_{t-1}^{k-1}, \mathbf{o}_{t-1}) & \text{if } k > 0 \end{cases} \qquad v_t^k = f_\theta^{value}(\mathbf{s}_t^k) \qquad r_t^k, \gamma_t^k = f_\theta^{out}(\mathbf{s}_t^{k-1}, \mathbf{o}_t).$$

Intuitively, $\mathbf{s}_t^k$ is the VPN's k-step prediction of the abstract-state at time t predicted from $\mathbf{x}_{t-k}$ following options $\mathbf{o}_{t-k}, ..., \mathbf{o}_{t-1}$ in the trajectory as illustrated in Figure 4.3. By applying the value and the outcome module, VPN can compute the k-step prediction of the value, the reward, and the discount. The k-step prediction loss at step t is defined as:

$$\mathcal{L}_t = \sum_{l=1}^{k} \left(R_t - v_t^l\right)^2 + \left(r_t - r_t^l\right)^2 + \left(\log_\gamma \gamma_t - \log_\gamma \gamma_t^l\right)^2$$

where $R_t = \begin{cases} r_t + \gamma_t R_{t+1} & \text{if } t \le n \\ \max_{\mathbf{o}} Q_{\theta^-}^d(\mathbf{s}_{n+1}, \mathbf{o}) & \text{if } t = n+1 \end{cases}$ is the target value, and $Q_{\theta^-}^d(\mathbf{s}_{n+1}, o)$ is the Q-value computed by the d-step planning method described in 4.3.2. Intuitively, $\mathcal{L}_t$ accumulates losses over 1-step to k-step predictions of values, rewards, and discounts. We find that applying $\log_\gamma$ for the discount prediction loss helps optimization, which amounts to computing the squared loss with respect to the number of steps.

Our learning algorithm introduces two hyperparameters: the number of prediction steps (k) and planning depth (d_{train}) used for choosing options and computing bootstrapped targets. We also make use of a *target network* parameterized by θ^- which is synchronized with θ after a certain number of steps to stabilize training as suggested by Mnih et al. (2016). The loss is accumulated over n-steps and the parameter is updated by computing its gradient as follows: $\nabla_\theta \mathcal{L} = \sum_{t=1}^{n} \nabla_\theta \mathcal{L}_t$.

4.3.4 Relationship to Existing Approaches

VPN is model-based in the sense that it learns an abstract-state transition function sufficient to predict rewards/discount/values. Meanwhile, VPN can also be viewed as model-free in the sense that it learns to directly estimate the value of the abstract-state. From this perspective, VPN exploits several auxiliary prediction tasks, such as reward and discount predictions to learn a good abstract-state representation. An interesting property of VPN is that its planning ability is used to compute the bootstrapped target as well as choose options during Q-learning. Therefore, as VPN improves the quality of its future predictions, it can not only perform better during evaluation through its improved planning ability, but also generate more accurate target Q-values during training, which encourages faster convergence compared to conventional Q-learning.

4.4 Experiments

Our experiments investigated the following questions: 1) Does VPN outperform model-free baselines (e.g., DQN)? 2) What is the advantage of planning with a VPN over observation-based planning? 3) Is VPN useful for complex domains with high-dimensional sensory inputs, such as Atari games?

4.4.1 Experimental Setting

Network Architecture. A CNN was used as the encoding module of VPN, and the transition module consists of one option-conditional convolution layer which uses different weights depending on the option followed by a few more convolution layers. We used a residual connection (He et al., 2016) from the previous abstract-state to the next abstract-state so that the transition module learns the change of the abstract-state. The outcome module is similar to the transition module except that it does not have a residual connection and two fully-connected layers are used to produce reward and discount. The value module consists of two fully-connected layers. The number of layers and hidden units vary depending on the domain.

Implementation Details. Our algorithm is based on asynchronous n-step Q-learning (Mnih et al., 2016) where n is 10 and 16 threads are used. The target network is synchronized after every 10K steps. We used the Adam optimizer (Kingma and Ba, 2015), and the best learning rate and its decay were chosen from $\{0.0001, 0.0002, 0.0005, 0.001\}$ and

$\{0.98, 0.95, 0.9, 0.8\}$ respectively. The learning rate is multiplied by the decay every 1M steps. Our implementation is based on TensorFlow (Abadi et al., 2016).[1]

VPN has four more hyperparameters: 1) the number of predictions steps (**k**) during training, 2) the plan depth ($\mathbf{d}_{train}$) during training, 3) the plan depth ($\mathbf{d}_{test}$) during evaluation, and 4) the branching factor (**b**) which indicates the number of options to be simulated for each expansion step during planning. We used $k = d_{train} = d_{test}$ throughout the experiment unless otherwise stated. **VPN(d)** represents our model which learns to predict and simulate up to d-step futures during training and evaluation. The branching factor (b) was set to 4 until depth of 3 and set to 1 after depth of 3, which means that VPN simulates 4-best options up to depth of 3 and only the best option after that.

Baselines. We compared our approach to the following baselines.

- **DQN**: This baseline directly estimates Q-values as its output and is trained through asynchronous n-step Q-learning. Unlike the original DQN, however, our DQN baseline takes an option as additional input and applies an option-conditional convolution layer to the top of the last encoding convolution layer, which is very similar to our VPN architecture.[2]

- **VPN(1)**: This is identical to our VPN with the same training procedure except that it performs only 1-step rollout to estimate Q-value as shown in Figure 4.1a. This can be viewed as a variation of DQN that predicts reward, discount, and the value of the next state as a decomposition of Q-value.

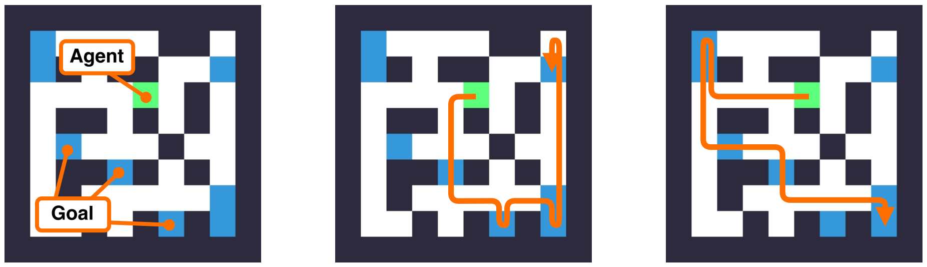
- **OPN(d)**: We call this Observation Prediction Network (OPN), which is similar to VPN except that it directly predicts future observations. More specifically, we train two independent networks: a model network ($f^{model} : \mathbf{x}, \mathbf{o} \mapsto r, \gamma, \mathbf{x}'$) which predicts reward, discount, and the next observation, and a value network ($f^{value} : \mathbf{x} \mapsto V(\mathbf{x})$) which estimates the value from the observation. The training scheme is similar to our algorithm except that a squared loss for observation prediction is used to train the model network. This baseline performs d-step planning like VPN(d).

4.4.2 Collect Domain

Task Description. We defined a simple but challenging 2D navigation task where the agent should collect as many goals as possible within a time limit, as illustrated in Figure 4.4.

[1]The code is available on `https://github.com/junhyukoh/value-prediction-network`.

[2]This architecture outperformed the original DQN architecture in our preliminary experiments.



(a) Observation (b) DQN's trajectory (c) VPN's trajectory

Figure 4.4: Collect domain. (a) The agent should collect as many goals as possible within a time limit which is given as additional input. (b-c) DQN collects 5 goals given 20 steps, while VPN(5) found the optimal trajectory via planning which collects 6 goals.

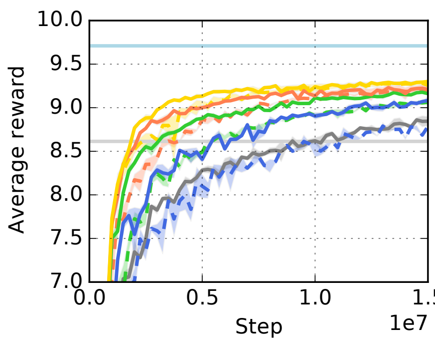
In this task, the agent, goals, and walls are randomly placed for each episode. The agent has four options: move left/right/up/down to the first crossing branch or the end of the corridor in the chosen direction. The agent is given 20 steps for each episode and receives a positive reward (2.0) when it collects a goal by moving on top of it and a time-penalty (-0.2) for each step. Although it is easy to learn a sub-optimal policy which collects nearby goals, finding the optimal trajectory in each episode requires careful planning because the optimal solution cannot be computed in polynomial time.

An observation is represented as a 3D tensor ($\mathbb{R}^{3\times10\times10}$) with binary values indicating the presence/absence of each object type. The time remaining is normalized to $[0, 1]$ and is concatenated to the 3rd convolution layer of the network as a channel.

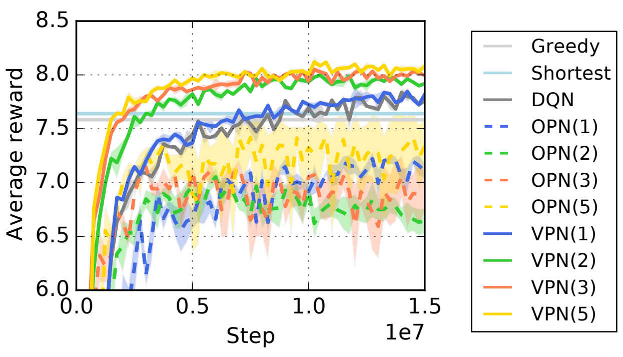
We evaluated all architectures first in a deterministic environment and then investigated the robustness in a stochastic environment separately. In the stochastic environment, each goal moves by one block with probability of 0.3 for each step. In addition, each option can be repeated multiple times with probability of 0.3. This makes it difficult to predict and plan the future precisely.

Overall Performance. The result is summarized in Figure 4.5. To understand the quality of different policies, we implemented a greedy algorithm which always collects the nearest goal first and a shortest-path algorithm which finds the optimal solution through exhaustive search assuming that the environment is deterministic. Note that even a small gap in terms of reward can be qualitatively substantial as indicated by the small gap between greedy and shortest-path algorithms.

The results show that many architectures learned a better-than-greedy policy in the deterministic and stochastic environments except that OPN baselines perform poorly in the stochastic environment. In addition, the performance of VPN is improved as the plan

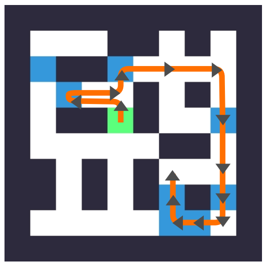


(a) Deterministic



(b) Stochastic

Figure 4.5: Learning curves on Collect domain. 'VPN(d)' represents VPN with d-step planning, while 'DQN' and 'OPN(d)' are the baselines.


(a) Plan with 20 steps

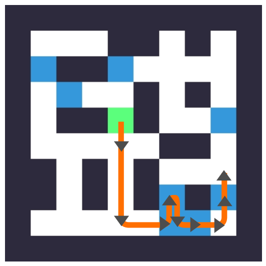

(b) Plan with 12 steps

Figure 4.6: Example of VPN's plan. VPN can plan the best future options just from the current state. The figures show VPN's different plans depending on the time limit.

depth increases, which implies that deeper predictions are reliable enough to provide more accurate value estimates of future states. As a result, VPN with 5-step planning represented by 'VPN(5)' performs best in both environments.

Comparison to Model-free Baselines. Our VPNs outperform DQN and VPN(1) baselines by a large margin as shown in Figure 4.5. Figure 4.4 (b-c) shows an example of trajectories of DQN and VPN(5) given the same initial state. Although DQN's behavior is reasonable, it ended up with collecting one less goal compared to VPN(5). We hypothesize that 6 convolution layers used by DQN and VPN(1) are not expressive enough to find the best route in each episode because finding an optimal path requires a combinatorial search in this task. On the other hand, VPN can perform such a combinatorial search to some extent by simulating future abstract-states, which has advantages over model-free approaches for dealing with tasks that require careful planning.

Table 4.1: Generalization performance. Each number represents average reward. 'FGs' and 'MWs' represent unseen environments with fewer goals and more walls respectively. Bold-faced numbers represent the highest rewards with 95% confidence level.

	Deterministic			Stochastic		
	Original	FGs	MWs	Original	FGs	MWs
Greedy	8.61	5.13	7.79	7.58	4.48	7.04
Shortest	9.71	5.82	8.98	7.64	4.36	7.22
DQN	8.66	4.57	7.08	7.85	4.11	6.72
VPN(1)	8.94	4.92	7.64	7.84	4.27	7.15
OPN(5)	**9.30**	**5.45**	**8.36**	7.55	4.09	6.79
VPN(5)	**9.29**	**5.43**	**8.31**	**8.11**	**4.45**	**7.46**

Comparison to Observation-based Planning. Compared to OPNs which perform planning based on predicted observations, VPNs perform slightly better or equally well in the deterministic environment. We observed that OPNs can predict future observations very accurately because observations in this task are simple and the environment is deterministic. Nevertheless, VPNs learn faster than OPNs in most cases. We conjecture that it takes additional training steps for OPNs to learn to predict future observations. In contrast, VPNs learn to predict only minimal but sufficient information for planning: reward, discount, and the value of future abstract-states, which may be the reason why VPNs learn faster than OPNs.

In the stochastic Collect domain, VPNs significantly outperform OPNs. We observed that OPNs tend to predict the average of possible future observations ($\mathbb{E}_{\mathbf{x}}[\mathbf{x}]$) because OPN is deterministic. Estimating values on such blurry predictions leads to estimating $V_\theta(\mathbb{E}_{\mathbf{x}}[\mathbf{x}])$ which is different from the true expected value $\mathbb{E}_{\mathbf{x}}[V(\mathbf{x})]$. On the other hand, VPN is trained to approximate the true expected value because there is no explicit constraint or loss for the predicted abstract state. We hypothesize that this key distinction allows VPN to learn different modes of possible future states more flexibly in the abstract state space. This result suggests that a value-prediction model can be more beneficial than an observation-prediction model when the environment is stochastic and building an accurate observation-prediction model is difficult.

Generalization Performance. One advantage of model-based RL approach is that it can generalize well to unseen environments as long as the dynamics of the environment remains similar. To see if our VPN has such a property, we evaluated all architectures on two types of previously unseen environments with either reduced number of goals (from 8 to 5) or

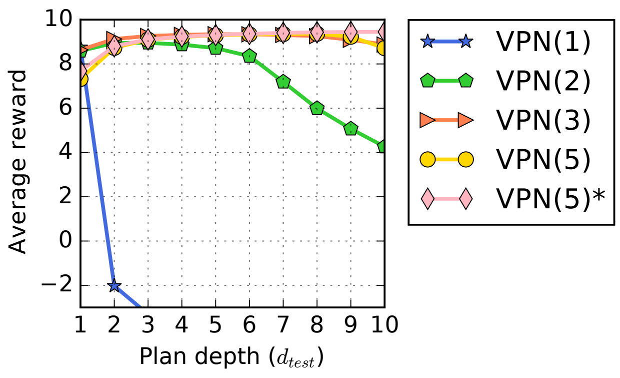


Figure 4.7: Effect of evaluation planning depth. Each curve shows average reward as a function of planning depth, d_{test}, for each architecture that is trained with a fixed number of prediction steps. 'VPN(5)*' was trained to make 10-step predictions but performed 5-step planning during training ($k = 10, d_{train} = 5$).

increased number of walls. It turns out that our VPN is much more robust to the unseen environments compared to model-free baselines (DQN and VPN(1)), as shown in Table 4.1. The model-free baselines perform worse than the greedy algorithm on unseen environments, whereas VPN still performs well. In addition, VPN generalizes as well as OPN which can learn a near-perfect model in the deterministic setting, and VPN significantly outperforms OPN in the stochastic setting. This suggests that VPN has a good generalization property like model-based RL methods and is robust to stochasticity.

Effect of Planning Depth. To further investigate the effect of planning depth in a VPN, we measured the average reward in the deterministic environment by varying the planning depth (d_{test}) from 1 to 10 during evaluation after training VPN with a fixed number of prediction steps and planning depth (k, d_{train}), as shown in Figure 4.7. Since VPN does not learn to predict observations, there is no guarantee that it can perform deeper planning during evaluation (d_{test}) than the planning depth used during training (d_{train}). Interestingly, however, the result in Figure 4.7 shows that if $k = d_{train} > 2$, VPN achieves better performance during evaluation through deeper tree search ($d_{test} > d_{train}$). We also tested a VPN with $k = 10$ and $d_{train} = 5$ and found that a planning depth of 10 achieved the best performance during evaluation. Thus, with a suitably large number of prediction steps during training, our VPN is able to benefit from deeper planning during evaluation relative to the planning depth during training. Figure 4.6 shows examples of good plans of length greater than 5 found by a VPN trained with planning depth 5. Another observation from Figure 4.7 is that the performance of planning depth of 1 ($d_{test} = 1$) degrades as the planning depth during training (d_{train}) increases. This means that a VPN can improve its value estimations

Table 4.2: Performance on Atari games. Each number represents average score over 5 top agents.

	Frostbite	Seaquest	Enduro	Alien	QBert	Ms. Pacman	Amidar	Krull	C. Climber
DQN	3058	2951	326	**1804**	12592	**2804**	535	12438	41658
VPN	**3811**	**5628**	**382**	1429	**14517**	2689	**641**	**15930**	**54119**

through long-term planning at the expense of the quality of short-term planning.

4.4.3 Atari Games

To investigate how VPN deals with complex visual observations, we evaluated it on several Atari games Bellemare et al. (2013). Unlike in the Collect domain, in Atari games most primitive actions have only small value consequences and it is difficult to hand-design useful extended options. Nevertheless, we explored if VPNs are useful in Atari games even with short-lookahead planning using simple options that repeat the same primitive action over extended time periods by using a frame-skip of 10.[3] We pre-processed the game screen to 84×84 gray-scale images. All architectures take last 4 frames as input. We doubled the number of hidden units of the fully-connected layer for DQN to approximately match the number of parameters. VPN learns to predict rewards and values but not discount (since it is fixed), and was trained to make 3-option-step predictions for planning which means that the agent predicts up to 0.5 seconds ahead in real-time.

As summarized in Table 4.2 and Figure 4.8, our VPN outperforms DQN baseline on 7 out of 9 Atari games and learned significantly faster than DQN on Seaquest, QBert, Krull, and Crazy Climber. One possible reason why VPN outperforms DQN is that even 3-step planning is indeed helpful for learning a better policy. Figure 4.9 shows an example of VPN's 3-step planning in Seaquest. Our VPN predicts reasonable values given different sequences of actions, which can potentially help choose a better action by looking at the short-term future. Another hypothesis is that the architecture of VPN itself, which has several auxiliary prediction tasks for multi-step future rewards and values, is useful for learning a good abstract-state representation as a model-free agent. Finally, our algorithm which performs planning to compute the target Q-value can potentially speed up learning by generating more accurate targets as it performs value backups multiple times from the simulated futures, as discussed in Section 4.3.4. These results show that our approach is applicable to complex visual environments without needing to predict observations.

[3]Much of the previous work on Atari games has used a frame-skip of 4. Though using a larger frame-skip generally makes training easier, it may make training harder in some games if they require more fine-grained control (Lakshminarayanan et al., 2017).

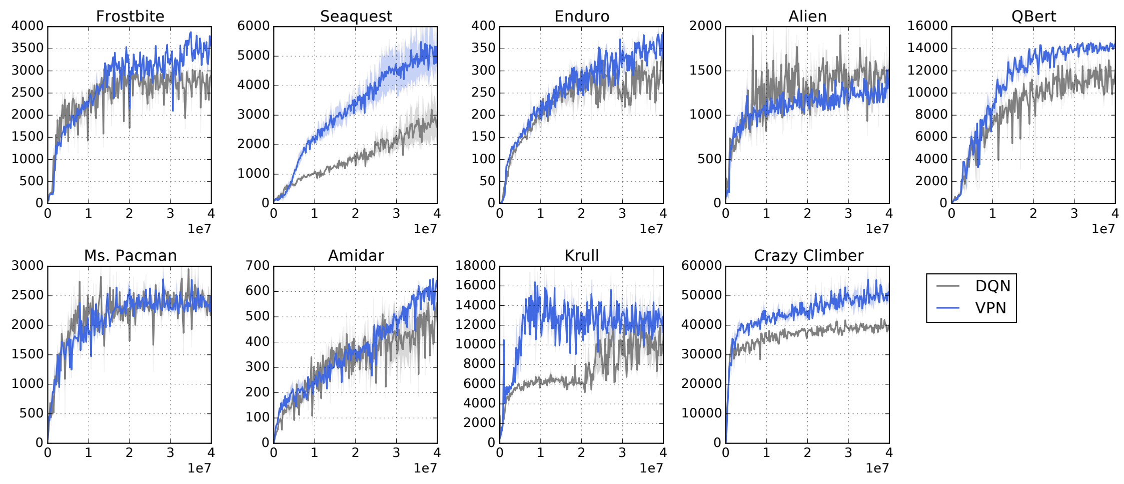
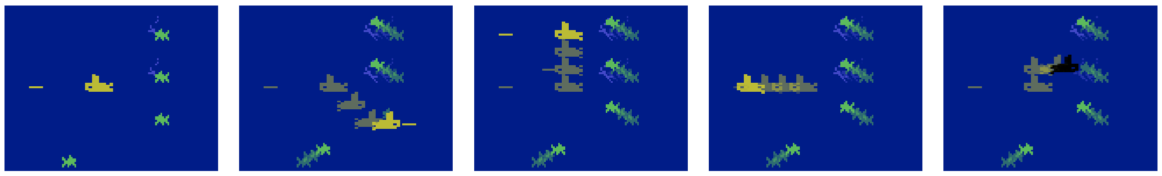


Figure 4.8: Learning curves on Atari games. X-axis and y-axis correspond to steps and average reward over 100 episodes respectively.



(a) State (b) Plan 1 (19.3) (c) Plan 2 (18.7) (d) Plan 3 (18.4) (e) Plan 4 (17.1)

Figure 4.9: Examples of VPN's value estimates. Each figure shows trajectories of different sequences of actions from the initial state (a) along with VPN's value estimates in the parentheses: $r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 V(s_4)$. The action sequences are (b) DownRight-DownRightFire-RightFire, (c) Up-Up-Up, (d) Left-Left-Left, and (e) Up-Right-Right. VPN predicts the highest value for (b) where the agent kills the enemy and the lowest value for (e) where the agent is killed by the enemy.

4.5 Discussion

We introduced value prediction networks (VPNs) as a new deep RL way of integrating planning and learning while simultaneously learning the dynamics of abstract-states that make option-conditional predictions of future rewards/discount/values rather than future observations. Our empirical evaluations showed that VPNs outperform model-free DQN baselines in multiple domains, and outperform traditional observation-based planning in a stochastic domain. An interesting future direction would be to develop methods that automatically learn the options that allow good planning in VPNs.

CHAPTER V

Neural Memory Architectures for Partially Observable Environment

In this chapter, we introduce a new set of reinforcement learning (RL) tasks in Minecraft (a flexible 3D world). We then use these tasks to systematically compare and contrast existing deep reinforcement learning (DRL) architectures with our new memory-based DRL architectures. These tasks are designed to emphasize, in a controllable manner, issues that pose challenges for RL methods including partial observability (due to first-person visual observations), delayed rewards, high-dimensional visual observations, and the need to use active perception in a correct manner so as to perform well in the tasks. While these tasks are conceptually simple to describe, by virtue of having all of these challenges simultaneously they are difficult for current DRL architectures. Additionally, we evaluate the generalization performance of the architectures on environments not used during training. The experimental results show that our new architectures generalize to unseen environments better than existing DRL architectures.

5.1 Introduction

Deep learning approaches (surveyed in LeCun et al., 2015; Schmidhuber, 2015) have made advances in many low-level perceptual supervised learning problems Krizhevsky et al. (2012); Girshick et al. (2014); Simonyan and Zisserman (2015). This success has been extended to reinforcement learning (RL) problems that involve visual perception. For example, the Deep Q-Network (DQN) Mnih et al. (2015) architecture has been shown to successfully learn to play many Atari 2600 games in the Arcade Learning Environment (ALE) benchmark Bellemare et al. (2013) by learning visual features useful for control directly from raw pixels using Q-Learning Watkins and Dayan (1992).

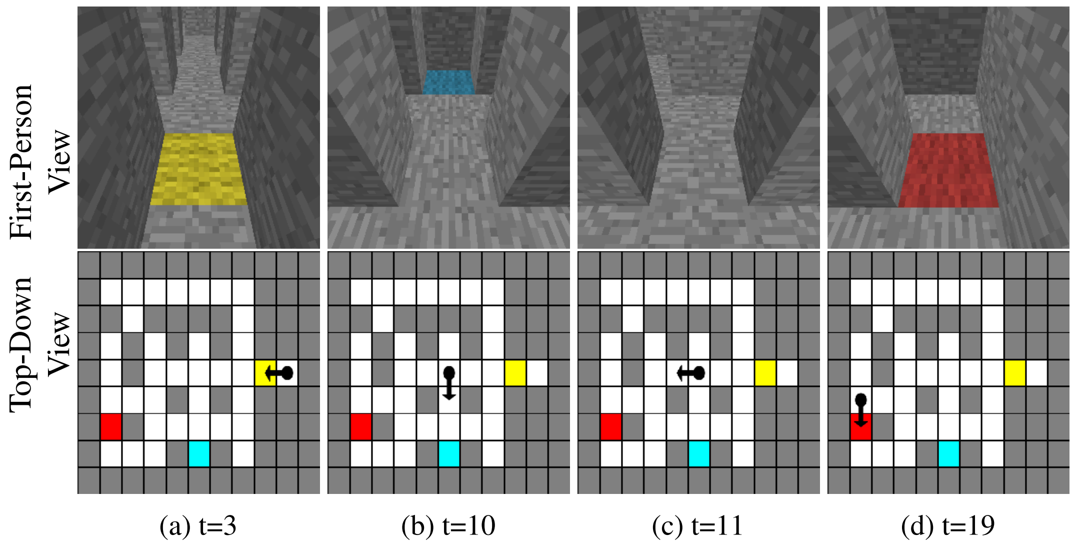


Figure 5.1: Example task in Minecraft. In this task, the agent should visit the red block if the indicator (next to the start location) is yellow. Otherwise, if the indicator is green, it should visit the blue block. The top row shows the agent's first-person observation. The bottom row visualizes the map and the agent's location; this is not available to the agent. (a) The agent observes the yellow indicator. (b) The agent looks left and sees the blue block, (c) but it decides to keep going straight having previously seen the yellow indicator. (d) Finally, it visits the red block and receives a positive reward.

Recently, researchers have explored problems that require faculties associated with higher-level cognition (e.g., inferring simple general purpose algorithms: Graves et al., 2014, and, Q&A: Weston et al., 2015). Most of these advances, however, are restricted to the supervised learning setting, which provides clear error signals. In this chapter, we are interested in extending this success to similarly cognition-inspired RL tasks. Specifically, this chapter introduces a set of tasks in *Minecraft*[1], a flexible 3D world in which an agent can collect resources, build structures, and survive attacks from enemies. Our RL tasks (one example is illustrated in Figure 5.1) not only have the usual RL challenges of partial observability, high-dimensional (visual) perception, and delayed reward, but also require an agent to develop movement policies by learning how to use its active perception to observe useful information and collect reward. In addition, our RL tasks require an agent to learn to use any memory it possesses including its interaction with active perception which feeds observations into memory. We note that for simplicity we hereafter refer to these cognition-inspired tasks as cognitive tasks but acknowledge that they form at best a very limited exploration of the range of cognitive faculties in humans.

In this work, we aim to not only systematically evaluate the performance of different

[1]https://minecraft.net/

neural network architectures on our tasks, but also examine how well such architectures generalize to unseen or larger topologies (Minecraft maps). The empirical results show that existing DRL architectures Mnih et al. (2015); Hausknecht and Stone (2015) perform worse on unseen or larger maps compared to training sets of maps, even though they perform reasonably well on the training maps. Motivated by the lack of generalization of existing architectures on our tasks, we also propose new memory-based DRL architectures. Our proposed architectures store recent observations into their memory and retrieve relevant memory based on the temporal context, whereas memory retrieval in existing architectures used in RL problems is not conditioned on the context. In summary, we show that our architectures outperform existing ones on most of the tasks as well as generalize better to unseen maps by exploiting their new memory mechanisms.

5.2 Related Work

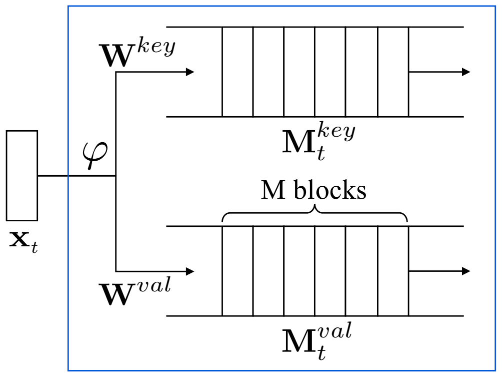
Neural Networks with External Memory. Graves et al. (2014) introduced a Neural Turing Machine (NTM), a differentiable external memory architecture, and showed that it can learn algorithms such as copy and reverse. Zaremba and Sutskever (2016) proposed RL-NTM that has a non-differentiable memory to scale up the addressing mechanism of NTM and applied policy gradient to train the architecture. Joulin and Mikolov (2015) implemented a stack using neural networks and demonstrated that it can infer several algorithmic patterns. Sukhbaatar et al. (2015b) proposed a Memory Network (MemNN) for Q&A and language modeling tasks, which stores all inputs and retrieves relevant memory blocks depending on the question.

Deep Reinforcement Learning. Neural networks have been used to learn features for RL tasks for a few decades (e.g., Tesauro, 1995 and Lange and Riedmiller, 2010). Recently, Mnih et al. (2015) proposed a Deep Q-Network (DQN) for training deep convolutional neural networks (CNNs) through Q-Learning in an end-to-end fashion; this achieved state-of-the-art performance on Atari games. Guo et al. (2014) used slow Monte-Carlo Tree Search (MCTS) Kocsis and Szepesvári (2006) to generate a relatively small amount of data to train fast-playing convolutional networks in Atari games. Schulman et al. (2015), Levine et al. (2016), and Lillicrap et al. (2016) have successfully trained deep neural networks to directly learn policies and applied their architectures to robotics problems. In addition, there are deep RL approaches to tasks other than Atari such as learning algorithms Zaremba et al. (2016) and text-based games Sukhbaatar et al. (2015a); Narasimhan et al. (2015). There have also been a few attempts to learn state-transition models using deep learning to improve exploration in RL Oh et al. (2015); Stadie et al. (2015). Most recently, Mnih et al. (2016) proposed asynchronous DQN and showed that it can learn to explore a 3D environment similar to Minecraft. Unlike their work, we focus on a systematic evaluation of the ability to deal with partial observability, active perception, and external memory in different neural network architectures as well as generalization across size and maps.

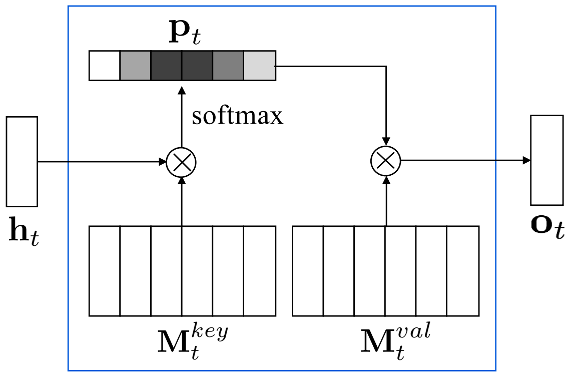
Model-free Deep RL for POMDPs. Building a model-free agent in partially observable Markov decision processes (POMDPs) is a challenging problem because the agent needs to learn how to summarize history for action-selection. To deal with such a challenge, Bakker et al. (2003) used a Long Short-Term Memory (LSTM) network Hochreiter and Schmidhuber (1997) in an offline policy learning framework to show that a robot controlled by an LSTM network can solve *T-Mazes* where the robot should go to the correct destination depending on the traffic signal at the beginning of the maze. Wierstra et al. (2010) proposed a *Recurrent Policy Gradient* method and showed that an LSTM network trained using this method outperforms other methods in several tasks including T-Mazes. More recently, Zhang et al. (2016) introduced continuous memory states to augment the state and action space and showed it can memorize salient information through *Guided Policy Search* Levine and Koltun (2013). Hausknecht and Stone (2015) proposed Deep Recurrent Q-Network (DRQN) which consists of an LSTM on top of a CNN based on the DQN framework and demonstrated improved handling of partial observability in Atari games.

Departure from Related Work. The architectures we introduce use memory mechanisms similar to MemNN, but our architectures have a layer that constructs a query for memory retrieval based on temporal context. Our architectures are also similar to NTM in that a recurrent controller interacts with an external memory, but ours have a simpler writing and addressing mechanism which makes them easier to train. Most importantly, our architectures are used in an RL setting and must learn from a delayed reward signal, whereas most previous work in exploring architectures with memory is in the supervised learning setting with its much more direct and undelayed error signals. We describe details of our architectures in Section 5.3.

The tasks we introduce are inspired by the T-maze experiments Bakker et al. (2003) as well as MazeBase Sukhbaatar et al. (2015a), which has natural language descriptions of mazes available to the agent. Unlike these previous tasks, our mazes have high-dimensional visual observations with deep partial observability due to the nature of the 3D worlds. In addition, the agent has to learn how best to control its active perception system to collect useful information at the right time in our tasks; this is not necessary in previous work.

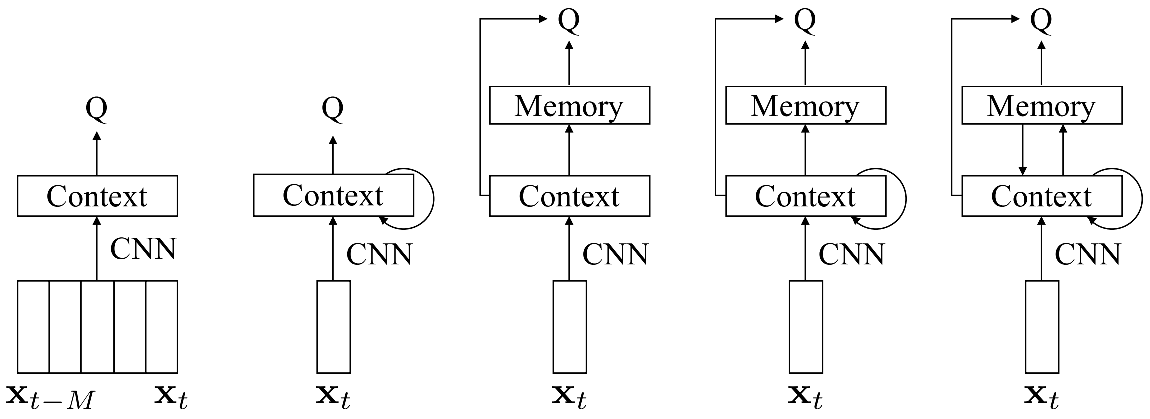


(a) Write



(b) Read

Figure 5.2: Illustration of memory operations.



(a) DQN (b) DRQN (c) MQN (d) RMQN (e) FRMQN

Figure 5.3: Illustration of different architectures

5.3 Architectures

The importance of retrieving a prior observation from memory depends on the current context. For example, in the maze of Figure 5.1 where the color of the indicator block determines the desired target color, the indicator information is important only when the agent is seeing a potential target and has to decide whether to approach it or find a different target. Motivated by the lack of "context-dependent memory retrieval" in existing DRL architectures, we present three new memory-based architectures in this section.

Our proposed architectures (Figure 5.3c-e) consist of convolutional networks for extracting high-level features from images (§5.3.1), a memory that retains a recent history of observations (§5.3.2), and a context vector used both for memory retrieval and (in part for) action-value estimation (§5.3.3). Depending on how the context vector is constructed, we obtain three new architectures: Memory Q-Network (MQN), Recurrent Memory Q-Network (RMQN), and Feedback Recurrent Memory Q-Network (FRMQN).

5.3.1 Encoding

For each time-step, a raw observation (pixels) is encoded to a fixed-length vector as follows:

$$\mathbf{e}_t = \varphi^{enc}(\mathbf{x}_t) \tag{5.1}$$

where $\mathbf{x}_t \in \mathbb{R}^{c \times h \times w}$ is $h \times w$ image with c channels, and $\mathbf{e}_t \in \mathbb{R}^e$ is the encoded feature at time t. In this work, we use a CNN to encode the observation.

5.3.2 Memory

The memory operations in the proposed architectures are similar to those proposed in MemNN.

Write. The encoded features of last M observations are linearly transformed and stored into the memory as *key* and *value* memory blocks as illustrated in Figure 5.2a. More formally, two types of memory blocks are defined as follows:

$$\mathbf{M}_t^{key} = \mathbf{W}^{key}\mathbf{E}_t \tag{5.2}$$

$$\mathbf{M}_t^{val} = \mathbf{W}^{val}\mathbf{E}_t \tag{5.3}$$

where $\mathbf{M}_t^{key}, \mathbf{M}_t^{val} \in \mathbb{R}^{m \times M}$ are memory blocks with m-dimensional embeddings, and $\mathbf{W}^{key}, \mathbf{W}^{val} \in \mathbb{R}^{m \times e}$ are parameters of the linear transformations for keys and values respectively. $\mathbf{E}_t = [\mathbf{e}_{t-1}, \mathbf{e}_{t-2}, ..., \mathbf{e}_{t-M}] \in \mathbb{R}^{e \times M}$ is the concatenation of features of the last M observations.

Read. The reading mechanism of the memory is based on soft attention Graves (2013); Bahdanau et al. (2015) as illustrated in Figure 5.2b. Given a context vector $\mathbf{h}_t \in \mathbb{R}^m$ (§5.3.3), the memory module draws soft attention over memory locations (and implicitly time) by computing the inner-product between the context and all key memory blocks as follows:

$$p_{t,i} = \frac{\exp\left(\mathbf{h}_t^\top \mathbf{M}_t^{key}[i]\right)}{\sum_{j=1}^{M} \exp\left(\mathbf{h}_t^\top \mathbf{M}_t^{key}[j]\right)} \tag{5.4}$$

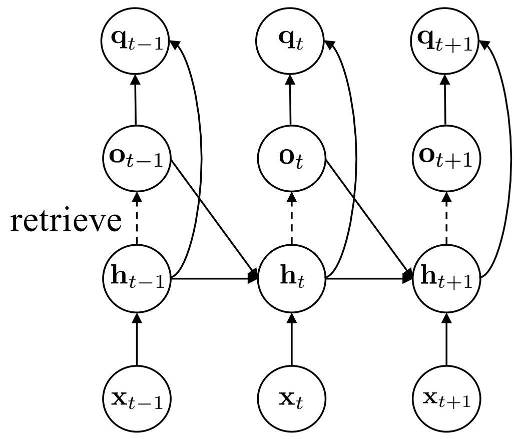


Figure 5.4: Unrolled illustration of FRMQN.

where $p_{t,i} \in \mathbb{R}$ is an attention weight for i-th memory block ($t - i$ time-step). The output of the read operation is the linear sum of the value memory blocks based on the attention weights as follows:

$$\mathbf{o}_t = \mathbf{M}_t^{val}\mathbf{p}_t \tag{5.5}$$

where $\mathbf{o}_t \in \mathbb{R}^m$ and $\mathbf{p}_t \in \mathbb{R}^M$ are the retrieved memory and the attention weights respectively.

5.3.3 Context

To retrieve useful information from memory, the context vector should capture relevant spatio-temporal information from the observations. To this end, we present three different architectures for constructing the context vector:

$$\text{MQN: } \mathbf{h}_t = \mathbf{W}^c\mathbf{e}_t \tag{5.6}$$

$$\text{RMQN: } [\mathbf{h}_t, \mathbf{c}_t] = \text{LSTM}\left(\mathbf{e}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}\right) \tag{5.7}$$

$$\text{FRMQN: } [\mathbf{h}_t, \mathbf{c}_t] = \text{LSTM}\left([\mathbf{e}_t, \mathbf{o}_{t-1}], \mathbf{h}_{t-1}, \mathbf{c}_{t-1}\right) \tag{5.8}$$

where $\mathbf{h}_t, \mathbf{c}_t \in \mathbb{R}^m$ are a context vector and a memory cell of LSTM respectively, and $[\mathbf{e}_t, \mathbf{o}_{t-1}]$ denotes concatenation of the two vectors as input for LSTM. **MQN** is a feedforward architecture that constructs the context based on only the current observation, which is very similar to MemNN except that the current input is used for memory retrieval in the temporal context of an RL problem. **RMQN** is a recurrent architecture that captures spatio-temporal information from the history of observations using LSTM. This architecture allows for retaining temporal information through LSTM as well as external memory. Finally,

FRMQN has a feedback connection from the retrieved memory to the context vector as illustrated in Figure 5.4. This allows the FRMQN architecture to refine its context based on the previously retrieved memory so that it can do more complex reasoning as time goes on. Note that feedback connections are analogous to the idea of *multiple hops* in MemNN in the sense that the architecture retrieves memory blocks multiple times based on the previously retrieved memory. However, FRMQN retrieves memory blocks through time, while MemNN does not.

Finally, the architectures estimate action-values by incorporating the retrieved memory and the context vector:

$$\mathbf{q}_t = \varphi^q \left(\mathbf{h}_t, \mathbf{o}_t\right) \tag{5.9}$$

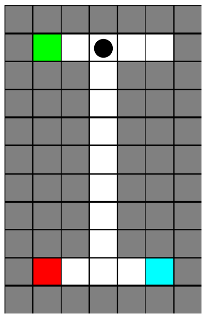
where $\mathbf{q}_t \in \mathbb{R}^a$ is the estimated action-value, and φ^q is a multi-layer perceptron (MLP) taking two inputs. In the results we report here, we used an MLP with one hidden layer as follows: $\mathbf{g}_t = f\left(\mathbf{W}^h\mathbf{h}_t + \mathbf{o}_t\right), \mathbf{q}_t = \mathbf{W}^q\mathbf{g}_t$ where f is a rectified linear function Nair and Hinton (2010) applied only to half of the hidden units for easy optimization by following Sukhbaatar et al. (2015b).

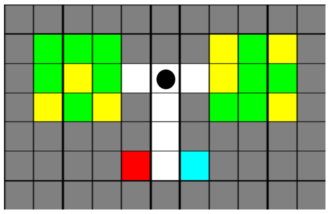
5.4 Experiments

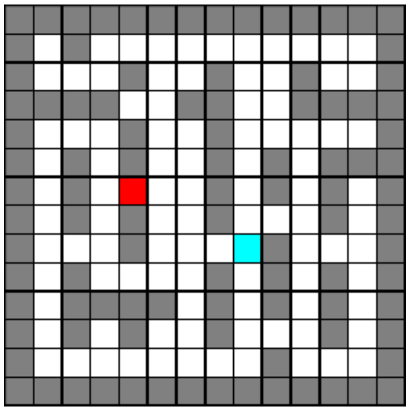
The experiments, baselines, and tasks are designed to investigate how useful context-dependent memory retrieval is for generalizing to unseen maps, and when memory feedback connections in FRMQN are helpful. Game play videos can be found in the following website: https://sites.google.com/a/umich.edu/junhyuk-oh/icml2016-minecraft. Next, we describe aspects that are common to all tasks and our training methodology.

Environment. In all the tasks, episodes terminate either when the agent finishes the task or after 50 steps. An agent receives -0.04 reward at every time step. The agent's initial looking direction is randomly selected among four directions: north, south, east, and west. For tasks where there is randomness (e.g., maps, spawn points), we randomly sampled an instance after every episode.

Actions. The following six actions are available: Look left/right ($\pm 90^\circ$ in yaw), Look up/down ($\pm 45^\circ$ in pitch), and Move forward/backward. Moving actions move the agent one block forward or backward in the direction it is facing. The pitch is limited to $[-45^\circ, 0^\circ]$.


(a) I-Maze


(b) Pattern Matching


(c) Random Maze

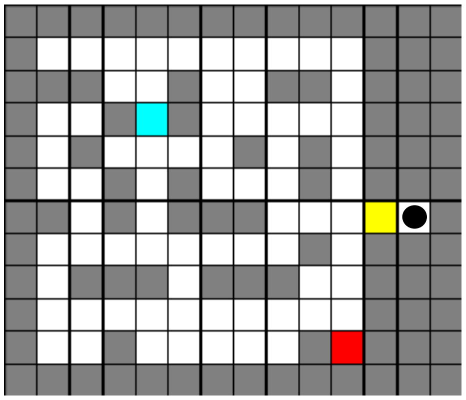

(d) Random Maze w/ Ind

Figure 5.5: Examples of maps. (a) has an I-structured topology where the location of indicator (yellow/green), goals (red/blue), and spawn locations (black circle) are fixed across episodes. (b) has two goals and two rooms with color patterns. (c) consists of randomly generated walls and two goals. The agent can be spawned anywhere except for goal locations. (d) is similar to (c) except that it has an indicator at the fixed location (yellow/green) and a fixed spawn location.

Baselines. We compare our three architectures with two baselines: **DQN** Mnih et al. (2015) (see Figure 5.3a) and **DRQN** Hausknecht and Stone (2015) (see Figure 5.3b). DQN is a CNN architecture that takes a fixed number of frames as input. DRQN is a recurrent architecture that has an LSTM layer on top of the CNN. Note that DQN cannot take more than the number of frames used during training because its first convolution layer takes a fixed number of observations. However, DRQN and our architectures can take arbitrary number of input frames using their recurrent layers. Additionally, our architectures can use an arbitrarily large size of memory during evaluation as well.

Training details. Input frames from Minecraft are captured as 32×32 RGB images. All the architectures use the same 2-layer CNN architecture. In the DQN and DRQN

Table 5.1: Performance on I-Maze. Each entry shows the average success rate with standard error measured from 10 runs. For each run, we measured the average success rate of 10 best-performing parameters based on the performance on unseen set of maps. The success rate is defined as the number of episodes that the agent reaches the correct goal within 100 steps divided by the total number of episodes. 'Size' represents the number of blocks of the vertical corridor. '✓' indicates that such sizes of I-Mazes belong to the training set of maps.

Size	Train	DQN	DRQN	MQN	RMQN	FRMQN
4		92.1(1.5)	94.8(1.5)	87.2(2.3)	89.2(2.4)	**96.9**(1.0)
5	✓	**99.3**(0.5)	98.2(1.1)	96.2(1.0)	98.6(0.5)	**99.3**(0.7)
6		**99.4**(0.4)	98.2(1.0)	96.0(1.0)	99.0(0.4)	**99.7**(0.3)
7	✓	99.6(0.3)	98.8(0.8)	98.0(0.6)	98.8(0.5)	**100.0**(0.0)
8		99.3(0.4)	98.3(0.8)	98.3(0.5)	98.0(0.8)	**100.0**(0.0)
9	✓	99.0(0.5)	98.4(0.6)	98.0(0.7)	94.6(1.8)	**100.0**(0.0)
10		96.5(0.7)	97.4(1.1)	98.2(0.7)	87.5(2.6)	**99.6**(0.3)
15		50.7(0.9)	83.3(3.2)	**96.7**(1.3)	89.8(2.4)	**97.4**(1.1)
20		48.3(1.0)	63.6(3.7)	97.2(0.9)	96.3(1.2)	**98.8**(0.5)
25		48.1(1.0)	57.6(3.7)	**98.2**(0.7)	90.3(2.5)	**98.4**(0.6)
30		48.6(1.0)	60.5(3.6)	**97.9**(0.9)	87.1(2.4)	**98.1**(0.6)
35		49.5(1.2)	59.0(3.4)	**95.0**(1.1)	84.0(3.2)	**94.8**(1.2)
40		46.6(1.2)	59.2(3.6)	77.2(4.2)	71.3(5.0)	**89.0**(2.6)

architectures, the last convolutional layer is followed by a fully-connected layer with 256 hidden units. In our architectures, the last convolution layer is given as the encoded feature for memory blocks. In addition, 256 LSTM units are used in DRQN, RMQN, and FRMQN. Our implementation is based on Torch7 Collobert et al. (2011), a public DQN implementation Mnih et al. (2015), and a Minecraft Forge Mod.[2]

5.4.1 I-Maze: Description and Results

Task. Our I-Maze task was inspired by T-Mazes which have been used in animal cognition experiments Olton (1979). Maps for this task (see Figure 5.5a) have an indicator at the top that has equal chance of being yellow or green. If the indicator is yellow, the red block gives +1 reward and the blue block gives -1 reward; if the indicator is green, the red block gives -1 and the blue block gives +1 reward. Thus, the agent should memorize the color of the indicator at the beginning while it is in view and visit the correct goal depending on the indicator-color. We varied the length of the vertical corridor to $l = \{5, 7, 9\}$ during training. The last 12 frames were given as input for all architectures, and the size of memory for our architectures was 11.

[2]http://files.minecraftforge.net/

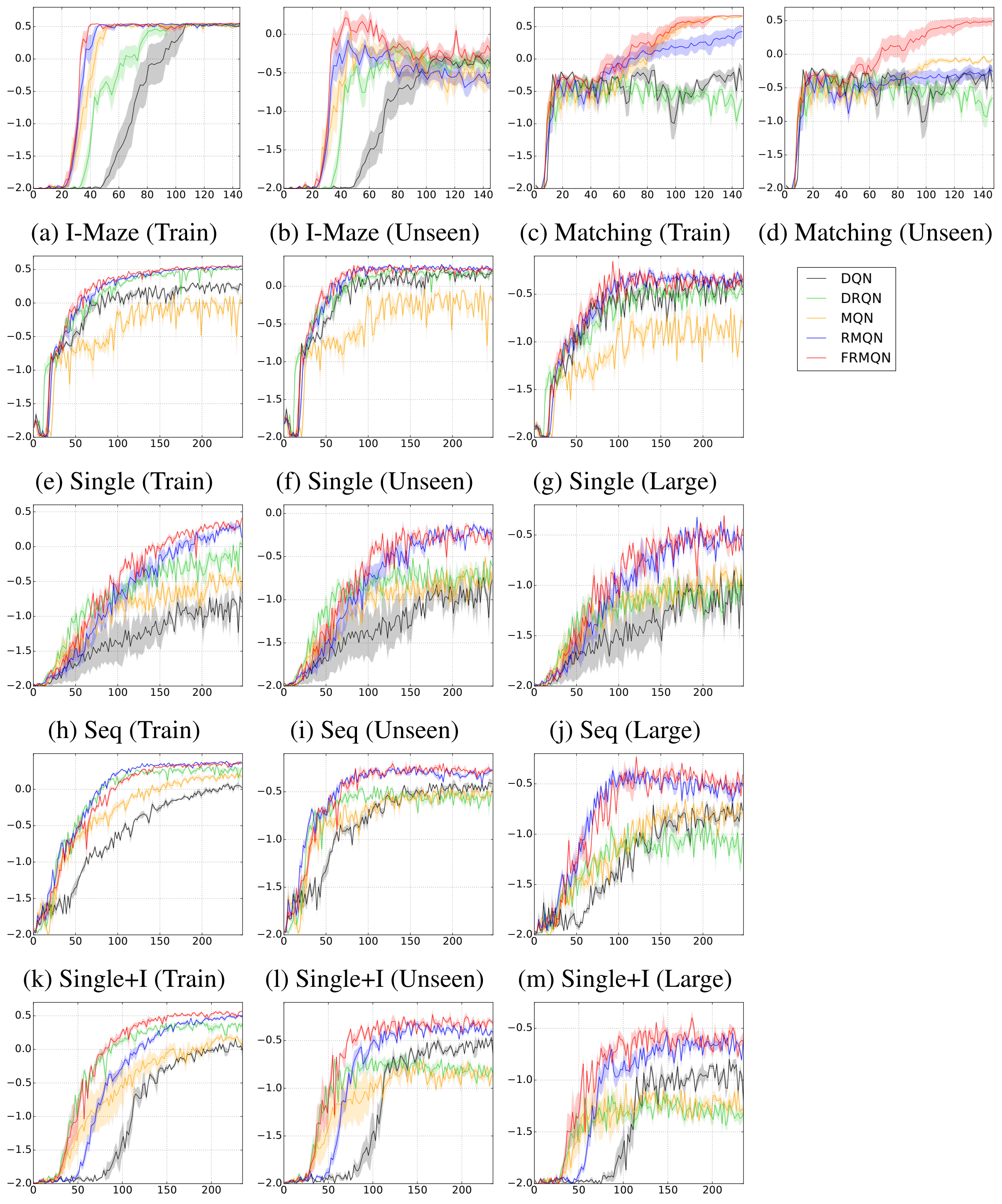


Figure 5.6: Learning curves for different tasks: (a-b) I-maze (§5.4.1), (c-d) pattern matching (§5.4.2), (e-p) random mazes (§5.4.3). X-axis and y-axis correspond to the number of training epochs (1 epoch = 10K steps) and the average reward. For (b) and (d), 'Unseen' represents unseen maps with different sizes and different patterns respectively. For random mazes, 'Unseen' and 'Large' indicate unseen topologies with the same sizes and larger sizes of maps, respectively. The performance was measured from 4 runs for random mazes and 10 runs for I-Maze and Pattern Matching.

Performance on the training set. We observed two stages of behavior during learning from all the architectures: 1) early in the training the discount factor and time penalty led to the agent to take a chance by visiting any goal, and 2) later in the training the agent goes to the correct goal by learning the correlation between the indicator and the goal. As seen in the learning curves in Figure 5.6a, our architectures converge more quickly than DQN and DRQN to the correct behavior. In particular, we observed that DRQN takes many more epochs to reach the second stage after the first stage has been reached. This is possibly due to the long time interval between seeing the indicator and the goals. Besides, the indicator block is important only when the agent is at the bottom end of the vertical corridor and needs to decide which way to go (see Figure 5.5a). In other words, the indicator information does not affect the agent's decision making along its way to the end of the corridor. This makes it even more difficult for DRQN to retain the indicator information for a long time. On the other hand, our architectures can handle these problems by storing the history of observations into memory and retrieving such information when it is important, based on the context.

Generalization performance. To investigate generalization performance, we evaluated the architectures on maps that have vertical corridor lengths $\{4, 6, 8, 10, 15, 20, 25, 30, 35, 40\}$ that were not present in the training maps. More specifically, testing on $\{6, 8\}$ sizes of maps and the rest of the sizes of maps can evaluate *interpolation* and *extrapolation* performance, respectively Schaul et al. (2015). Since some unseen maps are larger than the training maps, we used 50 last frames as input during evaluation on the unseen maps for all architectures except for DQN, which can take only 12 frames as discussed in the experimental setup. The size of memory for our architectures is set to 49. The performance on the unseen set of maps is visualized in Figure 5.6b. Although the generalization performances of all architectures are highly variable even after training performance converges, it can be seen that FRMQN consistently outperforms the other architectures in terms of average reward. To further investigate the performance for different lengths of the vertical corridor, we measured the performance on each size of map in Table 5.1. It turns out that all architectures perform well on $\{6, 8\}$ sizes of maps, which indicates that they can interpolate within the training set of maps. However, our architectures extrapolate to larger maps significantly better than the two baselines.

Analysis of memory retrieval. Figure 5.7a visualizes FRMQN's memory retrieval on a large I-Maze, where FRMQN sharply retrieves the indicator information only when it reaches the end of the corridor where it then makes a decision of which goal block to visit.

Table 5.2: Performance on pattern matching. The entries represent the probability of visiting the correct goal block for each set of maps with standard error. The performance reported is averages over 10 runs and 10 best-performing parameters for each run.

	Train	Unseen
DQN	62.9% (±3.4%)	60.1% (±2.8%)
DRQN	49.7% (±0.2%)	49.2% (±0.2%)
MQN	99.0% (±0.2%)	69.3% (±1.5%)
RMQN	82.5% (±2.5%)	62.3% (±1.5%)
FRMQN	**100.0%** (±0.0%)	**91.8%** (±1.0%)

This is a reasonable strategy because the indicator information is important only when it is at the end of the vertical corridor. This qualitative result implies that FRMQN learned a general strategy that looks for the indicator, goes to the end of the corridor, and retrieves the indicator information when it decides which goal block to visit. We observed similar policies learned by MQN and RMQN, but the memory attention for the indicator was not as sharp as FRMQN's attention and so they visit wrong goals in larger I-Mazes more often.

The results on I-Maze shown above suggest that solving a task on a set of maps does not guarantee solving the same task on similar but unseen maps, and such generalization performance highly depends on the feature representation learned by deep neural networks. The extrapolation result shows that context-dependent memory retrieval in our architectures is important for learning a general strategy when the importance of an observational-event depends highly on the temporal context.

5.4.2 Pattern Matching: Description and Results

Task. As illustrated in Figure 5.5b, this map consists of two 3×3 rooms. The visual patterns of the two rooms are either identical or different with equal probability. If the two rooms have the exact same color patterns, the agent should visit the blue block. If the rooms have different color patterns, the agent should visit the red block. The agent receives a +1 reward if it visits the correct block and a -1 reward if it visits the wrong block. This pattern matching task requires more complex reasoning (comparing two visual patterns given at different time steps) than the I-Maze task above. We generated 500 training and 500 unseen maps in such a way that there is little overlap between the two sets of visual patterns. The last 10 frames were given as input for all architectures, and the size of memory was set to 9.

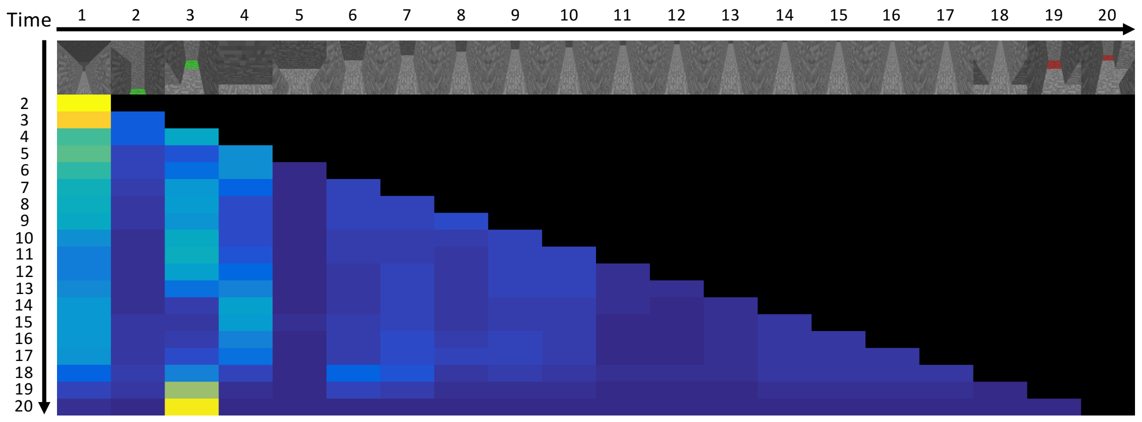
Performance on the training set. The results plotted in Figure 5.6c and Table 5.2 show that MQN and FRMQN successfully learned to go to the correct goal block for all runs in the training maps. We observed that DRQN always learned a sub-optimal policy that goes to any goal regardless of the visual patterns of the two rooms. Another observation is the training performances of DQN and RMQN are a bit unstable; they often learned the same sub-optimal policy, whereas MQN and FRMQN consistently learned to go to the correct goal across different runs. We hypothesize that it is not trivial for a neural network to compare two visual patterns observed in different time-steps unless the network can model high-order interactions between two specific observations for visual matching, which might be the reason why DQN and DRQN fail more often. Context-dependent memory retrieval mechanism in our architectures can alleviate this problem by retrieving two visual patterns corresponding to the observations of the two rooms before decision making.

Generalization performance. Table 5.2 and Figure 5.6d show that FRMQN achieves the highest success rate on the unseen set of maps. Interestingly, MQN fails to generalize to unseen visual patterns. We observed that MQN pays attention to the two visual patterns before choosing one of the goals through its memory retrieval. However, since the retrieved memory is just a convex combination of two visual patterns, it is hard for MQN to compare the similarity between them. Thus, we believe that MQN simply overfits to the training maps by memorizing the weighted sum of pairs of visual patterns in the training set of maps. On the other hand, FRMQN can utilize retrieved memory as well as its recurrent connections to compare visual patterns over time.

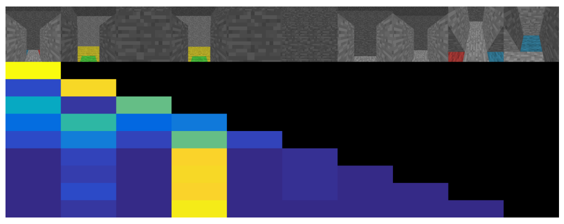
Analysis of memory retrieval. An example of FRMQN's memory retrieval is visualized in Figure 5.7b. FRMQN pays attention to both rooms, gradually moving weight from one to the other as time progresses, which means that the context vector is repeatedly refined based on the encoded features of the room retrieved through its feedback connections. Given this visualization and its good generalization performance, we hypothesize that FRMQN utilizes its feedback connection to compare the two visual features over time rather than comparing them at a single time-step. This result supports our view that feedback connections can play an important role in tasks where more complex reasoning is required with retrieved memories.

5.4.3 Random Mazes: Description and Results

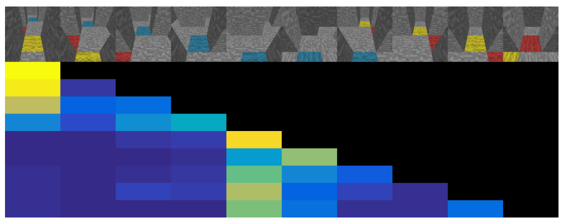
Task. A random maze task consists of randomly generated walls and goal locations as shown in Figure 5.5c and 5.5d. We present 4 classes of tasks using random mazes.



(a) I-Maze (§5.4.1)



(b) Pattern matching (§5.4.2)



(c) Sequential w/ Ind (§5.4.3)

Figure 5.7: Visualization of FRMQN's memory retrieval. Each figure shows a trajectory of FRMQN at the top row, and the following rows visualize attention weights over time. (a) The agent looks at the indicator, goes to the end of the corridor, and retrieves the indicator frame before visiting the goal block. (b) The agent looks at both rooms at the beginning and gradually switches attention weights from one room to another room as it approaches the goal blocks. (c) The agent pays attention to the indicator (yellow) and the first goal block (blue).

Table 5.3: Performance on random maze. The ‘Size’ column lists the size of each set of maps. The entries in the ‘Reward’, ‘Success’, and ‘Fail’ columns are average rewards, success rates, and failure rates measured from 4 runs. We picked the 10 best parameters based on performance on unseen maps for each run and evaluated them on 1000 episodes. ‘Success’ represents the number of correctly completed episodes divided by the total number of episodes, and ‘Fail’ represents the number of incorrectly completed episodes divided by the total number of episodes (e.g., visiting goals in reverse order in sequential goal tasks). The standard errors are lower than 0.03, 1.5%, 1.0% for all average rewards, success rates, and failure rates respectively.

Task	Type	Size	DQN			DRQN			MQN			RMQN			FRMQN		
			Reward	Success	Fail	Reward	Success	Fail	Reward	Success	Fail	Reward	Success	Fail	Reward	Success	Fail
Single	Train	4-8	0.31	90.4	0.6	**0.45**	94.5	0.1	0.01	78.8	0.4	**0.49**	95.7	0.1	**0.46**	94.6	0.3
	Unseen	4-8	0.22	87.3	0.7	0.23	86.6	0.2	0.02	79.4	0.3	**0.30**	89.4	0.3	**0.26**	88.0	0.5
	Large	9-14	**-0.28**	70.0	0.3	−0.40	63.0	0.1	−0.63	54.3	0.4	**-0.28**	69.3	0.1	**-0.28**	69.0	0.1
Seq	Train	5-7	−0.60	47.6	0.8	−0.08	66.0	0.6	−0.48	52.1	0.1	**0.21**	77.0	0.2	**0.22**	77.6	0.2
	Unseen	5-7	−0.66	45.0	1.0	−0.54	48.5	0.9	−0.59	48.4	0.1	**-0.13**	64.3	0.1	**-0.18**	63.1	0.3
	Large	8-10	−0.82	36.6	1.4	−0.89	32.6	1.0	−0.77	38.9	0.6	**-0.43**	49.6	1.1	**-0.42**	50.8	1.0
Single+I	Train	5-7	−0.04	79.3	6.3	0.23	87.9	1.2	0.11	83.9	0.7	**0.34**	91.7	0.8	0.24	88.0	1.4
	Unseen	5-7	−0.41	64.8	16.1	−0.46	61.0	13.4	−0.46	64.2	7.8	**-0.27**	70.0	10.2	**-0.23**	71.8	8.2
	Large	8-10	−0.74	49.4	31.6	−0.98	38.5	28.3	−0.66	55.5	17.1	**-0.39**	63.4	20.4	**-0.43**	63.4	17.2
Seq+I	Train	4-6	−0.13	68.0	7.0	0.25	78.5	1.1	−0.07	67.7	2.3	0.37	83.7	1.0	**0.48**	87.4	0.9
	Unseen	4-6	−0.58	54.5	14.5	−0.65	48.8	9.7	−0.71	47.3	7.2	**-0.32**	62.4	7.2	**-0.28**	63.8	7.5
	Large	7-9	−0.95	39.1	17.8	−1.14	30.2	13.1	−1.04	34.4	9.9	**-0.60**	49.5	12.5	**-0.54**	51.5	12.9

- **Single Goal**: The task is to visit the blue block which gives +1 reward while avoiding the red block that gives -1 reward.
- **Sequential Goals**: The task is to visit the red block first and then the blue block later which gives +0.5 and +1 reward respectively. If an agent visits the colored blocks in the reverse order, it receives -0.5 and -1 reward respectively.
- **Single Goal with Indicator**: If the indicator is yellow, the task is to visit the red block. If the indicator is green, the task is to visit the blue block. Visiting the correct block results in +1 reward and visiting the incorrect block results in -1 reward.
- **Sequential Goals with Indicator**: If the indicator is yellow, the task is to visit the blue block first and then the red block. If the indicator is green, the task is to visit the red block first and then the blue block. Visiting the blocks in the correct order results in +0.5 for the first block and +1 reward for the second block. Visiting the blocks in the reverse order results in -0.5 and -1 reward respectively.

We randomly generated 1000 maps used for training and two types of unseen evaluation sets of maps: 1000 maps of the same sizes present in the training maps and 1000 larger maps. The last 10 frames were given as input for all architectures, and the size of memory was set to 9.

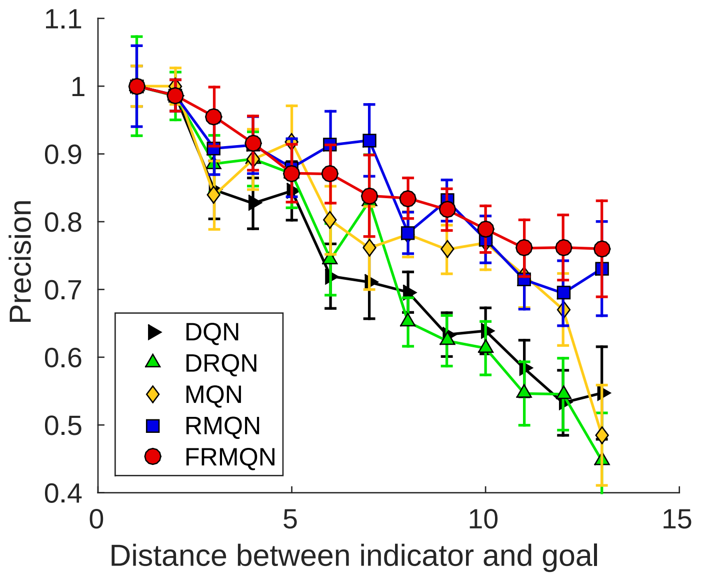


Figure 5.8: Precision vs. distance. X-axis represents the distance between indicator and goal in Single Goal with Indicator task. Y-axis represents the number of correct goal visits divided by the total number of goal visits.

Performance on the training set. In this task, the agent not only needs to remember important information while traversing the maps (e.g., an indicator) but it also has to search for the goals as different maps have different obstacle and goal locations. Table 5.3 shows that RMQN and FRMQN achieve higher asymptotic performances than the other architectures on the training set of maps.

Generalization performance. For the larger-sized unseen maps, we terminated episodes after 100 steps rather than 50 steps and used a time penalty of -0.02 considering their size. During evaluation, we used 10 frames as input for DQN and DRQN and 30 frames for MQN, RMQN, and FRMQN; these choices gave the best results for each architecture.

The results in Table 5.3 show that, as expected, the performance of all the architectures worsen in unseen maps. From the learning curves (see Figure 5.6e-g), we observed that generalization performance on unseen maps does not improve after some epochs, even though training performance is improving. This implies that improving policies on a fixed set of maps does not necessarily guarantee better performance on new environments. However, RMQN and FRMQN generalize better than the other architectures in most of the tasks. In particular, compared to the other architectures, DRQN's performance is significantly degraded on unseen maps. In addition, while DQN shows good generalization performance on the Single Goal task which primarily requires search, on the other tasks it tends to go to any goal regardless of important information (e.g., color of indicator). This can be seen through the higher failure rate (the number of incorrectly completed episodes divided by the total number of episodes) of DQN on indicator tasks in Table 5.3.

To investigate how well the architectures handle partial observability, we measured precision (proportion of correct goal visits to all goal visits) versus the distance between goal and indicator in Single Goal with Indicator task, which is visualized in Figure 5.8. Notably, the gap between our architectures (RMQN and FRMQN) and the other architectures becomes larger as the distance increases. This result implies that our architectures are better at handling partial observability than the other architectures, because large distance between indicator and goal is more likely to introduce deeper partial observability (i.e., long-term dependency).

Compared to MQN, the RMQN and FRMQN architectures achieve better generalization performance which suggests that the recurrent connections in the latter two architectures are a crucial component for handling random topologies. In addition, FRMQN and RMQN achieve similar performances, which implies that the feedback connection may not be always helpful in these tasks. We note that given a retrieved memory (e.g., indicator), the reasoning required for these tasks is simpler than the reasoning required for Pattern Matching task.

Analysis of memory retrieval. An example of memory retrieval in FRMQN is visualized in Figure 5.7c. It retrieves memory that contains important information (e.g., indicator) before it visits a goal block. The memory retrieval strategy is reasonable and is an evidence that the proposed architectures make it easier to generalize to large-scale environments by better handling partial observability.

5.5 Discussion

In this chapter, we introduced three classes of cognition-inspired tasks in Minecraft and compared the performance of two existing architectures with three architectures that we proposed here. We emphasize that unlike most evaluations of RL algorithms, we trained and evaluated architectures on disjoint sets of maps so as to specifically consider the applicability of learned value functions to unseen (interpolation and extrapolation) maps.

In summary, our main empirical result is that context-dependent memory retrieval, particularly with a feedback connection from the retrieved memory, can more effectively solve our set of tasks that require control of active perception and external physical movement actions. Our architectures, particularly FRQMN, also show superior ability relative to the baseline architectures when learning value functions whose behavior generalizes better from training to unseen environments. In future work, we intend to take advantage of the flexibility of the Minecraft domain to construct even more challenging cognitive tasks to further evaluate our architectures.

CHAPTER VI

Neural Hierarchical Architectures for Zero-Shot Task Generalization

As a step towards developing zero-shot task generalization capabilities in reinforcement learning (RL), this chapter introduces a new RL problem where the agent should learn to execute sequences of instructions after learning useful skills that solve subtasks. In this problem, we consider two types of generalizations: to previously unseen instructions and to longer sequences of instructions. For generalization over unseen instructions, we propose a new objective which encourages learning correspondences between similar subtasks by making analogies. For generalization over sequential instructions, we present a hierarchical architecture where a meta controller learns to use the acquired skills for executing the instructions. To deal with delayed reward, we propose a new neural architecture in the meta controller that learns when to update the subtask, which makes learning more efficient. Experimental results on a stochastic 3D domain show that the proposed ideas are crucial for generalization to longer instructions as well as unseen instructions.

6.1 Introduction

The ability to understand and follow instructions allows us to perform a large number of new complex sequential tasks even without additional learning. For example, we can make a new dish following a recipe, and explore a new city following a guidebook. Developing the ability to execute instructions can potentially allow reinforcement learning (RL) agents to generalize quickly over tasks for which such instructions are available. For example, factory-trained household robots could execute novel tasks in a new house following a human user's instructions (e.g., tasks involving household chores, going to a new place, picking up/manipulating new objects, etc.). In addition to generalization over instructions,

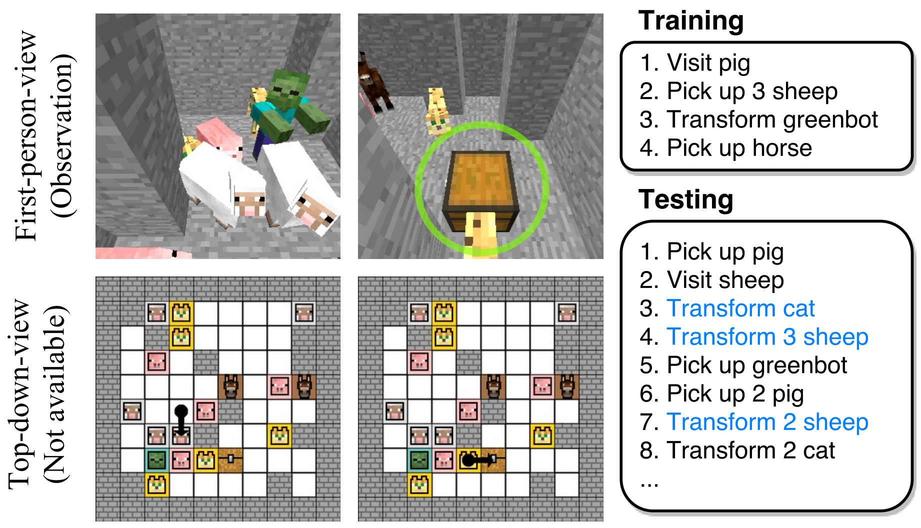


Figure 6.1: Example of 3D world and instructions. The agent is tasked to execute longer sequences of instructions in the correct order after training on short sequences of instructions; in addition, previously unseen instructions can be given during evaluation (blue text). Additional reward is available from randomly appearing boxes regardless of instructions (green circle).

an intelligent agent should also be able to handle unexpected events (e.g., low battery, arrivals of reward sources) while executing instructions. Thus, the agent should not blindly execute instructions sequentially but sometimes deviate from instructions depending on circumstances, which requires balancing between two different objectives.

Problem. To develop such capabilities, this chapter introduces the instruction execution problem where the agent's overall task is to execute a given list of instructions described by a simple form of natural language while dealing with unexpected events, as illustrated in Figure 6.1. More specifically, we assume that each instruction can be executed by performing one or more high-level subtasks in sequence. Even though the agent can pre-learn skills to perform such subtasks (e.g., [Pick up, Pig] in Figure 6.1), and the instructions can be easily translated to subtasks, our problem is difficult due to the following challenges.

- *Generalization*: Pre-training of skills can only be done on a subset of subtasks, but the agent is required to perform previously unseen subtasks (e.g., going to a new place) in order to execute unseen instructions during testing. Thus, the agent should learn to generalize to new subtasks in the skill learning stage. Furthermore, the agent is required to execute previously unseen and longer sequences of instructions during evaluation.
- *Delayed reward*: The agent is *not* told which instruction to execute at any point of time from the environment but just given the full list of instructions as input. In addition,

the agent does *not* receive any signal on completing individual instructions from the environment, i.e., success-reward is provided only when all instructions are executed correctly. Therefore, the agent should keep track of which instruction it is executing and decide when to move on to the next instruction.

- *Interruption*: As described in Figure 6.1, there can be unexpected events in uncertain environments, such as opportunities to earn bonuses (e.g., windfalls), or emergencies (e.g., low battery). It can be better for the agent to *interrupt* the ongoing subtask before it is finished, perform a different subtask to deal with such events, and resume executing the interrupted subtask in the instructions after that. Thus, the agent should achieve a balance between executing instructions and dealing with such events.
- *Memory*: There are loop instructions (e.g., "Pick up 3 pig") which require the agent to perform the same subtask ([Pick up, Pig]) multiple times and take into account the history of observations and subtasks in order to decide when to move on to the next instruction correctly.

Due to these challenges, the agent should be able to execute a novel subtask, keep track of what it has done, monitor observations to interrupt ongoing subtasks depending on circumstances, and switch to the next instruction precisely when the current instruction is finished.

Our Approach and Technical Contributions. To address the aforementioned challenges, we divide the learning problem into two stages: 1) learning skills to perform a set of subtasks and generalizing to unseen subtasks, and 2) learning to execute instructions using the learned skills. Specifically, we assume that subtasks are defined by several disentangled parameters. Thus, in the first stage our architecture learns a *parameterized skill* (da Silva et al., 2012) to perform different subtasks depending on input parameters. In order to generalize over unseen parameters, we propose a new objective function that encourages making analogies between similar subtasks so that the underlying manifold of the entire subtask space can be learned without experiencing all subtasks. In the second stage, our architecture learns a meta controller on top of the parameterized skill so that it can read instructions and decide which subtask to perform. The overall hierarchical RL architecture is shown in Figure 6.3. To deal with delayed reward as well as interruption, we propose a novel neural network (see Figure 6.4) that learns when to update the subtask in the meta controller. This not only allows learning to be more efficient under delayed reward by operating at a larger time-scale but also allows interruptions of ongoing subtasks when an unexpected event is observed.

Main Results. We developed a 3D visual environment using Minecraft based on Oh et al. (2016) where the agent can interact with many objects. Our results on multiple sets of parameterized subtasks show that our proposed analogy-making objective can generalize successfully. Our results on multiple instruction execution problems show that our meta controller's ability to learn when to update the subtask plays a key role in solving the overall problem and outperforms several hierarchical baselines. The demo videos are available at the following website: `https://sites.google.com/a/umich.edu/junhyuk-oh/task-generalization`.

The rest of the sections are organized as follows. Section 6.2 presents related work. Section 6.3 presents our analogy-making objective for generalization to parameterized tasks and demonstrates its application to different generalization scenarios. Section 6.4 presents our hierarchical architecture for the instruction execution problem with our new neural network that learns to operate at a large time-scale. In addition, we demonstrate our agent's ability to generalize over sequences of instructions, as well as provide a comparison to several alternative approaches.

6.2 Related Work

Hierarchical RL. A number of hierarchical RL approaches are designed to deal with sequential tasks. Typically these have the form of a meta controller and a set of lower-level controllers for subtasks (Sutton et al., 1999b; Dietterich, 2000; Parr and Russell, 1997; Ghavamzadeh and Mahadevan, 2003; Konidaris et al., 2012; Konidaris and Barto, 2007). However, much of the previous work assumes that the overall task is fixed (e.g., *Taxi* domain (Dietterich, 2000)). In other words, the optimal sequence of subtasks is fixed during evaluation (e.g., picking up a passenger followed by navigating to a destination in the Taxi domain). This makes it hard to evaluate the agent's ability to compose pre-learned policies to solve previously unseen sequential tasks in a zero-shot fashion unless we re-train the agent on the new tasks in a transfer learning setting (Singh, 1991, 1992; McGovern and Barto, 2002). Our work is also closely related to Programmable HAMs (PHAMs) (Andre and Russell, 2000, 2002) in that a PHAM is designed to execute a given program. However, the program explicitly specifies the policy in PHAMs which effectively reduces the state-action search space. In contrast, instructions are a description of the task in our work, which means that the agent should learn to use the instructions to maximize its reward.

Hierarchical Deep RL. Hierarhical RL has been recently combined with deep learning. Kulkarni et al. (2016a) proposed hierarchical Deep Q-Learning and demonstrated improved

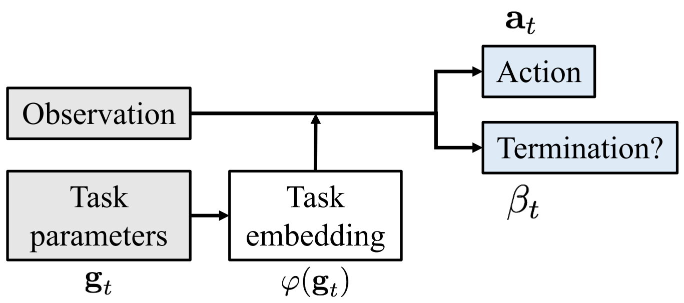


Figure 6.2: Architecture of parameterized skill. See text for details.

exploration in a challenging Atari game. Tessler et al. (2017) proposed a similar architecture, but the high-level controller is allowed to choose primitive actions directly. Bacon et al. (2017) proposed the *option-critic* architecture, which learns options without pseudo reward and demonstrated that it can learn distinct options in Atari games. Heess et al. (2016) formulated the actions of the meta controller as continuous variables that are used to modulate the behavior of the low-level controller. Florensa et al. (2017) trained a stochastic neural network with mutual information regularization to discover skills. Most of these approaches build an *open-loop* policy at the high-level controller that waits until the previous subtask is finished once it is chosen. This approach is not able to interrupt ongoing subtasks in principle, while our architecture can switch its subtask at any time.

Zero-Shot Task Generalization. There have been a few papers on zero-shot generalization to new tasks. For example, da Silva et al. (2012) introduced parameterized skills that map sets of task descriptions to policies. Isele et al. (2016) achieved zero-shot task generalization through dictionary learning with sparsity constraints. Schaul et al. (2015) proposed *universal value function approximators* (UVFAs) that learn value functions for state and goal pairs. Devin et al. (2017) proposed composing sub-networks that are shared across tasks and robots in order to achieve generalization to unseen configurations of them. Unlike the above prior work, we propose a flexible metric learning method which can be applied to various generalization scenarios. Andreas et al. (2017) proposed a framework to learn the underlying subtasks from a *policy sketch* which specifies a sequence of subtasks, and the agent can generalize over new sequences of them in principle. In contrast, our work aims to generalize over unseen subtasks as well as unseen sequences of them. In addition, the agent should handle unexpected events in our problem that are not described by the instructions by interrupting subtasks appropriately.

Instruction Execution. There has been a line of work for building agents that can execute natural language instructions: Tellex et al. (2011, 2014) for robotics and MacMahon et al.

(2006); Chen and Mooney (2011); Mei et al. (2016) for a simulated environment. However, these approaches focus on natural language understanding to map instructions to actions or *groundings* in a supervised setting. In contrast, we focus on generalization to sequences of instructions without any supervision for language understanding or for actions. Although Branavan et al. (2009) also tackle a similar problem, the agent is given a single instruction at a time, while our agent needs to learn how to align instructions and state given a full list of instructions.

6.3 Learning a Parameterized Skill

In this chapter, a parameterized skill is a multi-task policy corresponding to multiple tasks defined by categorical input *task parameters*, e.g., [Pick up, X]. More formally, we define a parameterized skill as a mapping $\mathcal{O} \times \mathcal{G} \rightarrow \mathcal{A} \times \mathcal{B}$, where $\mathcal{O}$ is a set of observations, $\mathcal{G}$ is a set of task parameters, $\mathcal{A}$ is a set of primitive actions, and $\mathcal{B} = \{0, 1\}$ indicates whether the task is finished or not. A space of tasks is defined using the Cartesian product of task parameters: $\mathcal{G} = \mathcal{G}^{(1)} \times \ldots \times \mathcal{G}^{(n)}$, where $\mathcal{G}^{(i)}$ is a set of the i-th parameters (e.g., $\mathcal{G} = \{\text{Visit, Pick up}\} \times \{\text{X, Y, Z}\}$). Given an observation $\mathbf{x}_t \in \mathcal{O}$ at time t and task parameters $\mathbf{g} = \left[g^{(1)}, \ldots, g^{(n)}\right] \in \mathcal{G}$, where $g^{(i)}$ is a one-hot vector, the parameterized skill is the following functions:

$$\text{Policy: } \pi_\phi(\mathbf{a}_t|\mathbf{x}_t, \mathbf{g})$$
$$\text{Termination: } \beta_\phi(b_t|\mathbf{x}_t, \mathbf{g}),$$

where π_ϕ is the policy optimized for the task $\mathbf{g}$, and β_ϕ is a termination function (Sutton et al., 1999b) which is the probability that the state is terminal at time t for the given task $\mathbf{g}$. The parameterized skill is represented by a non-linear function approximator $\phi(\cdot)$, a neural network in this chapter. The neural network architecture of our parameterized skill is illustrated in Figure 6.2. The network maps input task parameters into a task embedding space $\varphi(\mathbf{g})$, which is combined with the observation followed by the output layers.

6.3.1 Learning to Generalize by Analogy-Making

Only a subset of tasks ($\mathcal{G}' \subset \mathcal{G}$) are available during training, and so in order to generalize to unseen tasks during evaluation the network needs to learn knowledge about the relationship between different task parameters when learning the task embedding $\varphi(\mathbf{g})$.

To this end, we propose an analogy-making objective inspired by Reed et al. (2015).

The main idea is to learn correspondences between tasks. For example, if target objects and 'Visit/Pick up' actions are *independent* (i.e., each action can be applied to any target object), we can enforce the analogy [Visit, X] : [Visit, Y] :: [Pick up, X] : [Pick up, Y] for any X and Y in the embedding space, which means that the difference between 'Visit' and 'Pick up' is consistent regardless of target objects and vice versa. This allows the agent to generalize to unseen combinations of actions and target objects, such as performing [Pick up, Y] after it has learned to perform [Pick up, X] and [Visit, Y].

More specifically, we define several constraints as follows:

$$\begin{aligned} \|\Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D)\| &\approx 0 & \text{if } \mathbf{g}_A : \mathbf{g}_B :: \mathbf{g}_C : \mathbf{g}_D \\ \|\Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D)\| &\geq \tau_{dis} & \text{if } \mathbf{g}_A : \mathbf{g}_B \neq \mathbf{g}_C : \mathbf{g}_D \\ \|\Delta(\mathbf{g}_A, \mathbf{g}_B)\| &\geq \tau_{diff} & \text{if } \mathbf{g}_A \neq \mathbf{g}_B, \end{aligned}$$

where $\mathbf{g}_k = \left[g_k^{(1)}, g_k^{(2)}, ..., g_k^{(n)}\right] \in \mathcal{G}$ are task parameters, $\Delta(\mathbf{g}_A, \mathbf{g}_B) = \varphi(\mathbf{g}_A) - \varphi(\mathbf{g}_B)$ is the difference vector between two tasks in the embedding space, and τ_{dis} and τ_{diff} are constant threshold distances. Intuitively, the first constraint enforces the analogy (i.e., *parallelogram* structure in the embedding space; see Mikolov et al. (2013); Reed et al. (2015)), while the other constraints prevent trivial solutions. We incorporate these constraints into the following objectives based on contrastive loss (Hadsell et al., 2006):

$$\begin{aligned} \mathcal{L}_{sim} &= \mathbb{E}_{\mathbf{g}_{A...D} \sim \mathcal{G}_{sim}} \left[\|\Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D)\|^2\right] \\ \mathcal{L}_{dis} &= \mathbb{E}_{\mathbf{g}_{A...D} \sim \mathcal{G}_{dis}} \left[(\tau_{dis} - \|\Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D)\|)_+^2\right] \\ \mathcal{L}_{diff} &= \mathbb{E}_{\mathbf{g}_{A,B} \sim \mathcal{G}_{diff}} \left[(\tau_{diff} - \|\Delta(\mathbf{g}_A, \mathbf{g}_B)\|)_+^2\right], \end{aligned}$$

where $(\cdot)_+ = \max(0, \cdot)$ and $\mathcal{G}_{sim}, \mathcal{G}_{dis}, \mathcal{G}_{diff}$ are sets of task parameters that satisfy corresponding conditions in the above three constraints. The final analogy-making objective is the weighted sum of the above three objectives.

6.3.2 Training

The parameterized skill is trained on a set of tasks ($\mathcal{G}' \subset \mathcal{G}$) through the actor-critic method with generalized advantage estimation (Schulman et al., 2016). We also found that pre-training through *policy distillation* (Rusu et al., 2016; Parisotto et al., 2016) gives slightly better results as discussed in Tessler et al. (2017). Throughout training, the parameterized skill is also made to predict whether the current state is terminal or not through a binary classification objective, and the analogy-making objective is applied to the task embedding

Scenario	Analogy	Train	Unseen
Independent	×	**0.3** (99.8%)	-3.7 (34.8%)
	✓	**0.3** (99.8%)	**0.3** (99.5%)
Object-dependent	×	**0.3** (99.7%)	-5.0 (2.2%)
	✓	**0.3** (99.8%)	**0.3** (99.7%)
Inter/Extrapolation	×	**-0.7** (97.5%)	-2.2 (24.9%)
	✓	**-0.7** (97.5%)	**-1.7** (94.5%)

Table 6.1: Performance on parameterized tasks. Each entry shows 'Average reward (Success rate)'. We assume an episode is successful only if the agent successfully finishes the task and its termination predictions are correct throughout the whole episode.

separately.

6.3.3 Experiments

Environment. We developed a 3D visual environment using Minecraft based on Oh et al. (2016) as shown in Figure 6.1. An observation is represented as a 64×64 pixel RGB image. There are 7 different types of objects: *Pig*, *Sheep*, *Greenbot*, *Horse*, *Cat*, *Box*, and *Ice*. The topology of the world and the objects are randomly generated for every episode. The agent has 9 actions: *Look* (Left/Right/Up/Down), *Move* (Forward/Backward), *Pick up*, *Transform*, and *No operation*. *Pick up* removes the object in front of the agent, and *Transform* changes the object in front of the agent to ice (a special object).

Implementation Details. The network architecture of the parameterized skill consists of 4 convolution layers and one LSTM (Hochreiter and Schmidhuber, 1997) layer. We conducted curriculum training by changing the size of the world, the density of object and walls according to the agent's success rate. We implemented actor-critic method with 16 CPU threads based on Sukhbaatar et al. (2015a). The parameters are updated after 8 episodes for each thread.

Results. To see how useful analogy-making is for generalization to unseen parameterized tasks, we trained and evaluated the parameterized skill on three different sets of parameterized tasks defined below.

- **Independent**: The task space is defined as $\mathcal{G} = \mathcal{T} \times \mathcal{X}$, where $\mathcal{T} = \{\text{Visit}, \text{Pick up}, \text{Transform}\}$ and $\mathcal{X}$ is the set of object types. The agent should move on top of the target object given 'Visit' task and perform the corresponding actions in front of the target given 'Pick up' and 'Transform' tasks. Only a subset of tasks are encountered during training, so the agent should generalize over unseen configurations of task parameters.

- **Object-dependent**: The task space is defined as $\mathcal{G} = \mathcal{T}' \times \mathcal{X}$, where $\mathcal{T}' = \mathcal{T} \cup \{\text{Interact with}\}$. We divided objects into two groups, each of which should be either picked up or transformed given 'Interact with' task. Only a subset of target object types are encountered during training, so there is no chance for the agent to generalize without knowledge of the group of each object. We applied analogy-making so that analogies can be made only within the same group. This allows the agent to perform object-dependent actions even for unseen objects.
- **Interpolation/Extrapolation**: The task space is defined as $\mathcal{G} = \mathcal{T} \times \mathcal{X} \times \mathcal{C}$, where $\mathcal{C} = \{1, 2, ..., 7\}$. The agent should perform a task for a given number of times ($c \in \mathcal{C}$). Only $\{1, 3, 5\} \subset \mathcal{C}$ is given during training, and the agent should generalize over unseen numbers $\{2, 4, 6, 7\}$. Note that the optimal policy for a task can be derived from $\mathcal{T} \times \mathcal{X}$, but predicting termination requires generalization to unseen numbers. We applied analogy-making based on arithmetic (e.g., [Pick up, X, 2] : [Pick up, X, 5] :: [Transform, Y, 3] : [Transform, Y, 6]).

As summarized in Table 6.1, the parameterized skill with our analogy-making objective can successfully generalize to unseen tasks in all generalization scenarios. This suggests that when learning a representation of task parameters, it is possible to inject prior knowledge in the form of the analogy-making objective so that the agent can learn to generalize over unseen tasks in various ways depending on semantics or context without needing to experience them.

6.4 Learning to Execute Instructions using Parameterized Skill

We now consider the instruction execution problem where the agent is given a sequence of simple natural language instructions, as illustrated in Figure 6.1. We assume an already trained parameterized skill, as described in Section 6.3. Thus, the main remaining problem is how to use the parameterized skill to execute instructions. Although the requirement that instructions be executed sequentially makes the problem easier (than, e.g., conditional-instructions), the agent still needs to make complex decisions because it should deviate from instructions to deal with unexpected events (e.g., low battery) and remember what it has done to deal with loop instructions, as discussed in Section 6.1.

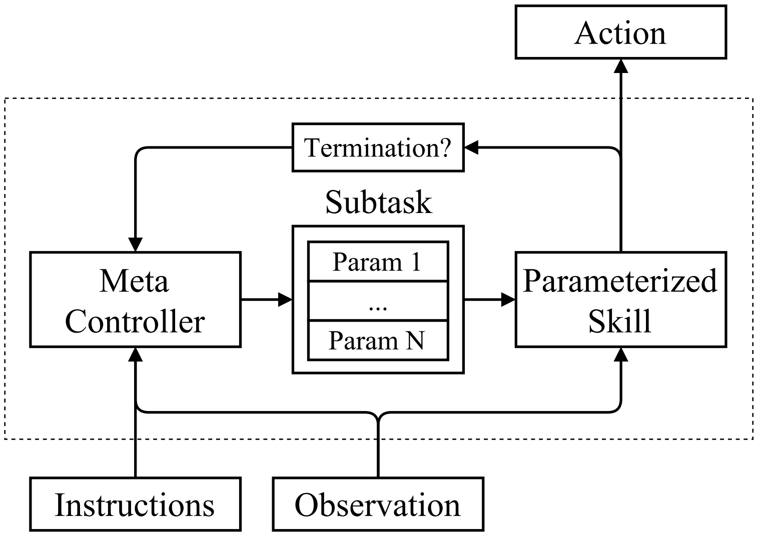


Figure 6.3: Overview of our hierarchical architecture.

To address the above challenges, our hierarchical RL architecture (see Figure 6.3) consists of two modules: meta controller and parameterized skill. Specifically, a meta controller reads the instructions and passes subtask parameters to a parameterized skill which executes the given subtask and provides its termination signal back to the meta controller. Section 6.4.1 describes the overall architecture of the meta controller for dealing with instructions. Section 6.4.2 describes a novel neural architecture that learns when to update the subtask in order to better deal with delayed reward signal as well as unexpected events.

6.4.1 Meta Controller Architecture

As illustrated in Figure 6.4, the meta controller is a mapping $\mathcal{O} \times \mathcal{M} \times \mathcal{G} \times \mathcal{B} \rightarrow \mathcal{G}$, where $\mathcal{M}$ is a list of instructions. Intuitively, the meta controller decides subtask parameters $\mathbf{g}_t \in \mathcal{G}$ conditioned on the observation $\mathbf{x}_t \in \mathcal{O}$, the list of instructions $M \in \mathcal{M}$, the previously selected subtask $\mathbf{g}_{t-1}$, and its termination signal ($b \sim \beta_\phi$).

In contrast to recent hierarchical deep RL approaches where the meta controller can update its subtask (or option) only when the previous one terminates or only after a fixed number of steps, our meta controller can update the subtask at any time and takes the termination signal as additional input. This gives more flexibility to the meta controller and enables interrupting ongoing tasks before termination.

In order to keep track of the agent's progress on instruction execution, the meta controller maintains its internal state by computing a *context* vector (Section 6.4.1.1) and determines which subtask to execute by focusing on one instruction at a time from the list of instructions

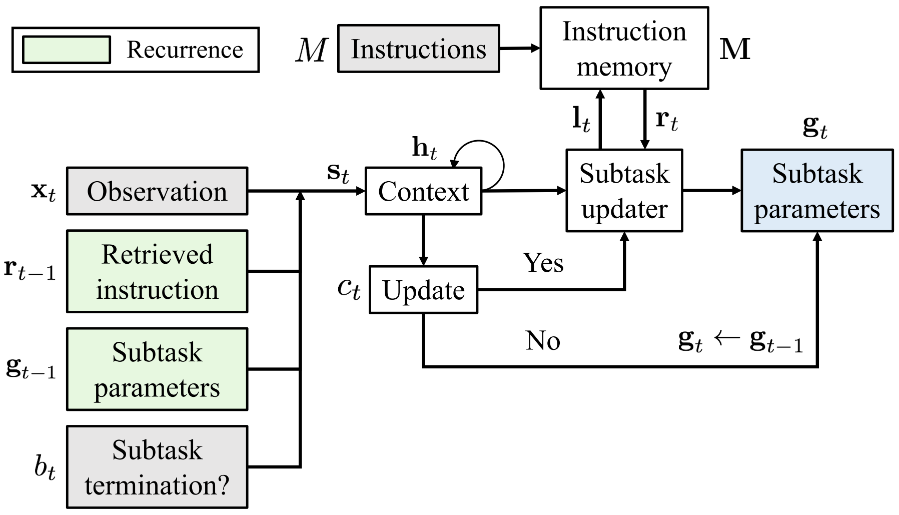


Figure 6.4: Neural network architecture of meta controller.

(Section 6.4.1.2).

6.4.1.1 Context

Given the sentence embedding $\mathbf{r}_{t-1}$ retrieved at the previous time-step from the instructions (described in Section 6.4.1.2), the previously selected subtask $\mathbf{g}_{t-1}$, and the subtask termination $b_t \sim \beta_\phi \left(b_t | \mathbf{s}_t, \mathbf{g}_{t-1}\right)$, the meta controller computes the context vector ($\mathbf{h}_t$) as follows:

$$\mathbf{h}_t = \text{LSTM}\left(\mathbf{s}_t, \mathbf{h}_{t-1}\right)$$
$$\mathbf{s}_t = f\left(\mathbf{x}_t, \mathbf{r}_{t-1}, \mathbf{g}_{t-1}, b_t\right),$$

where f is a neural network. Intuitively, $\mathbf{g}_{t-1}$ and b_t provide information about which subtask was being solved by the parameterized skill and whether it has finished or not. Thus, $\mathbf{s}_t$ is a summary of the current observation and the ongoing subtask. $\mathbf{h}_t$ takes the history of $\mathbf{s}_t$ into account through the LSTM, which is used by the subtask updater.

6.4.1.2 Subtask Updater

The *subtask updater* constructs a memory structure from the list of instructions, retrieves an instruction by maintaining a pointer into the memory, and computes the subtask parameters.

Instruction Memory. Given instructions as a list of sentences $M = (\mathbf{m}_1, \mathbf{m}_2, ..., \mathbf{m}_K)$, where each sentence consists of a list of words, $\mathbf{m}_i = \left(w_1, ..., w_{|\mathbf{m}_i|}\right)$, the subtask updater constructs memory blocks $\mathbf{M} \in \mathbb{R}^{E \times K}$ (i.e., each column is an E-dimensional embedding of a sentence). The subtask updater maintains an *instruction pointer* ($\mathbf{p}_t \in \mathbb{R}^{K}$) which is non-negative and sums up to 1 indicating which instruction the meta controller is executing. Memory construction and retrieval can be written as:

$$\text{Memory: } \mathbf{M} = \left[\varphi^{w}\left(\mathbf{m}_1\right), \varphi^{w}\left(\mathbf{m}_2\right), ..., \varphi^{w}\left(\mathbf{m}_K\right)\right] \tag{6.1}$$

$$\text{Retrieval: } \mathbf{r}_t = \mathbf{M}\mathbf{p}_t, \tag{6.2}$$

where $\varphi^{w}\left(\mathbf{m}_i\right) \in \mathbb{R}^{E}$ is the embedding of the i-th sentence (e.g., Bag-of-words), and $\mathbf{r}_t \in \mathbb{R}^{E}$ is the retrieved sentence embedding which is used for computing the subtask parameters. Intuitively, if $\mathbf{p}_t$ is a one-hot vector, $\mathbf{r}_t$ indicates a single instruction from the whole list of instructions. The meta controller should learn to manage $\mathbf{p}_t$ so that it can focus on the correct instruction at each time-step.

Since instructions should be executed sequentially, we use a location-based memory addressing mechanism (Zaremba and Sutskever, 2016; Graves et al., 2014) to manage the instruction pointer. Specifically, the subtask updater shifts the instruction pointer by $[-1, 1]$ as follows:

$$\mathbf{p}_t = \mathbf{l}_t * \mathbf{p}_{t-1} \text{ where } \mathbf{l}_t = \text{Softmax}\left(\varphi^{shift}(\mathbf{h}_t)\right), \tag{6.3}$$

where $*$ is a convolution operator, φ^{shift} is a neural network, and $\mathbf{l}_t \in \mathbb{R}^{3}$ is a soft-attention vector over the three shift operations $\{-1, 0, +1\}$. The optimal policy should keep the instruction pointer unchanged while executing an instruction and increase the pointer by +1 precisely when the current instruction is finished.

Subtask Parameters. The subtask updater takes the context ($\mathbf{h}_t$), updates the instruction pointer ($\mathbf{p}_t$), retrieves an instruction ($\mathbf{r}_t$), and computes subtask parameters as:

$$\pi_\theta\left(\mathbf{g}_t | \mathbf{h}_t, \mathbf{r}_t\right) = \prod_i \pi_\theta\left(g_t^{(i)} | \mathbf{h}_t, \mathbf{r}_t\right), \tag{6.4}$$

where $\pi_\theta\left(g_t^{(i)} | \mathbf{h}_t, \mathbf{r}_t\right) \propto \exp\left(\varphi_i^{goal}\left(\mathbf{h}_t, \mathbf{r}_t\right)\right)$, and φ_i^{goal} is a neural network for the i-th subtask parameter.

6.4.2 Learning to Operate at a Large Time-Scale

Although the meta controller can learn an optimal policy by updating the subtask at each time-step in principle, making a decision at every time-step can be inefficient because subtasks

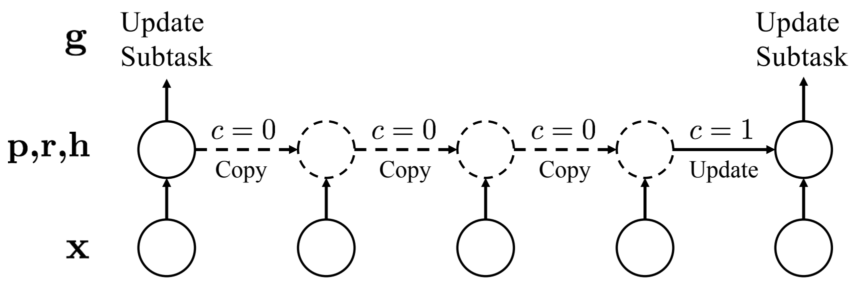


Figure 6.5: Unrolled illustration of the meta controller with a learned time-scale. The internal states ($\mathbf{p}, \mathbf{r}, \mathbf{h}$) and the subtask ($\mathbf{g}$) are updated only when $c = 1$. If $c = 0$, the meta controller continues the previous subtask without updating its internal states.

do not change frequently. Instead, having temporally-extended actions can be useful for dealing with delayed reward by operating at a larger time-scale (Sutton et al., 1999b). While it is reasonable to use the subtask termination signal to define the temporal scale of the meta controller as in many recent hierarchical deep RL approaches (see Section 6.2), this approach would result in a mostly open-loop meta-controller policy that is not able to interrupt ongoing subtasks before termination, which is necessary to deal with unexpected events not specified in the instructions.

To address this dilemma, we propose to learn the time-scale of the meta controller by introducing an internal binary decision which indicates whether to *invoke* the subtask updater to update the subtask or not, as illustrated in Figure 6.5. This decision is defined as: $c_t \sim \sigma\left(\varphi^{update}\left(\mathbf{s}_t, \mathbf{h}_{t-1}\right)\right)$ where σ is a sigmoid function. If $c_t = 0$, the meta controller continues the current subtask without updating the subtask updater. Otherwise, if $c_t = 1$, the subtask updater updates its internal states (e.g., instruction pointer) and the subtask parameters. This allows the subtask updater to operate at a large time-scale because one decision made by the subtask updater results in multiple actions depending on c values. The overall meta controller architecture with this update scheme is illustrated in Figure 6.4.

Soft-Update. To ease optimization of the non-differentiable variable (c_t), we propose a *soft-update* rule by using $c_t = \sigma\left(\varphi^{update}\left(\mathbf{s}_t, \mathbf{h}_{t-1}\right)\right)$ instead of sampling it. The key idea is to take the weighted sum of both 'update' and 'copy' scenarios using c_t as the weight. This method is described in Algorithm 5. We found that training the meta controller using soft-update followed by fine-tuning by sampling c_t is crucial for training the meta controller. Note that the soft-update rule reduces to the original formulation if we sample c_t and $\mathbf{l}_t$ from the Bernoulli and multinomial distributions, which justifies our initialization trick.

Algorithm 5 Subtask update (Soft)

```
**Input:** $\mathbf{s}_t, \mathbf{h}_{t-1}, \mathbf{p}_{t-1}, \mathbf{r}_{t-1}, \mathbf{g}_{t-1}$
**Output:** $\mathbf{h}_t, \mathbf{p}_t, \mathbf{r}_t, \mathbf{g}_t$
$c_t \leftarrow \sigma\left(\varphi^{update}\left(\mathbf{s}_t, \mathbf{h}_{t-1}\right)\right)$    # Decide update weight
$\tilde{\mathbf{h}}_t \leftarrow \text{LSTM}\left(\mathbf{s}_t, \mathbf{h}_{t-1}\right)$    # Update the context
$\mathbf{l}_t \leftarrow \text{Softmax}\left(\varphi^{shift}\left(\tilde{\mathbf{h}}_t\right)\right)$    # Decide shift operation
$\tilde{\mathbf{p}}_t \leftarrow \mathbf{l}_t * \mathbf{p}_{t-1}$    # Shift the instruction pointer
$\tilde{\mathbf{r}}_t \leftarrow \mathbf{M}\tilde{\mathbf{p}}_t$    # Retrieve instruction
# Merge two scenarios (update/copy) using $c_t$ as weight
$[\mathbf{p}_t, \mathbf{r}_t, \mathbf{h}_t] \leftarrow c_t[\tilde{\mathbf{p}}_t, \tilde{\mathbf{r}}_t, \tilde{\mathbf{h}}_t] + (1 - c_t)\left[\mathbf{p}_{t-1}, \mathbf{r}_{t-1}, \mathbf{h}_{t-1}\right]$
$g_t^{(i)} \sim c_t \pi_\theta\left(g_t^{(i)} | \tilde{\mathbf{h}}_t, \tilde{\mathbf{r}}_t\right) + (1 - c_t)\, g_{t-1}^{(i)} \forall i$
```

Integrating with Hierarchical RNN. The idea of learning the time-scale of a recurrent neural network is closely related to hierarchical RNN approaches (Koutnik et al., 2014; Chung et al., 2017) where different groups of recurrent hidden units operate at different time-scales to capture both long-term and short-term temporal information. Our idea can be naturally integrated with hierarchical RNNs by applying the update decision (c value) only for a subset of recurrent units instead of all the units. Specifically, we divide the context vector into two groups: $\mathbf{h}_t = \left[\mathbf{h}_t^{(l)}, \mathbf{h}_t^{(h)}\right]$. The low-level units ($\mathbf{h}_t^{(l)}$) are updated at every time-step, while the high-level units ($\mathbf{h}_t^{(h)}$) are updated depending on the value of c. This simple modification leads to a form of hierarchical RNN where the low-level units focus on short-term temporal information while the high-level units capture long-term dependencies.

6.4.3 Training

The meta controller is trained on a training set of lists of instructions. Given a pre-trained and fixed parameterized skill, the actor-critic method is used to update the parameters of the meta controller. Since the meta controller also learns a subtask embedding $\varphi(\mathbf{g}_{t-1})$ and has to deal with unseen subtasks during evaluation, analogy-making objective is also applied.

6.4.4 Experiments

The experiments are designed to explore the following questions: (1) Will the proposed hierarchical architecture outperform a non-hierarchical baseline? (2) How beneficial is the meta controller's ability to learn when to update the subtask? We are also interested in understanding the qualitative properties of our agent's behavior.

	Train	Test (Seen)	Test (Unseen)
Length of instructions	4	20	20
Flat	-7.1 (1%)	-63.6 (0%)	-62.0 (0%)
Hierarchical-Long	-5.8 (31%)	-59.2 (0%)	-59.2 (0%)
Hierarchical-Short	-3.3 (83%)	-53.4 (23%)	-53.6 (18%)
Hierarchical-Dynamic	**-3.1** (95%)	**-30.3** (75%)	**-38.0** (56%)

Table 6.2: Performance on instruction execution. Each entry shows average reward and success rate. 'Hierarchical-Dynamic' is our approach that learns when to update the subtask. An episode is successful only when the agent solves all instructions correctly.

Environment. We used the same Minecraft domain used in Section 6.3.3. The agent receives a time penalty (-0.1) for each step and receives $+1$ reward when it finishes the entire list of instructions in the correct order. Throughout an episode, a box (including treasures) randomly appears with probability of 0.03 and transforming a box gives $+0.9$ reward.

The subtask space is defined as $\mathcal{G} = \mathcal{T} \times \mathcal{X}$, and the semantics of each subtask are the same as the 'Independent' case in Section 6.3.3. We used the best-performing parameterized skill throughout this experiment.

There are 7 types of instructions: {Visit X, Pick up X, Transform X, Pick up 2 X, Transform 2 X, Pick up 3 X, Transform 3 X} where 'X' is the target object type. Note that the parameterized skill used in this experiment was not trained on loop instructions (e.g., Pick up 3 X), so the last four instructions require the meta controller to learn to repeat the corresponding subtask for the given number of times. To see how the agent generalizes to previously unseen instructions, only a subset of instructions and subtasks was presented during training.

Implementation Details. The meta controller consists of 3 convolution layers and one LSTM layer. We also conducted curriculum training by changing the size of the world, the density of object and walls, and the number of instructions according to the agent's success rate. We used the actor-critic implementation described in Section 6.3.3.

Baselines. To understand the advantage of using the hierarchical structure and the benefit of our meta controller's ability to learn when to update the subtask, we trained three baselines as follows.

- **Flat**: identical to our meta controller except that it directly chooses primitive actions without using the parameterized skill. It is also pre-trained on the training set of subtasks.

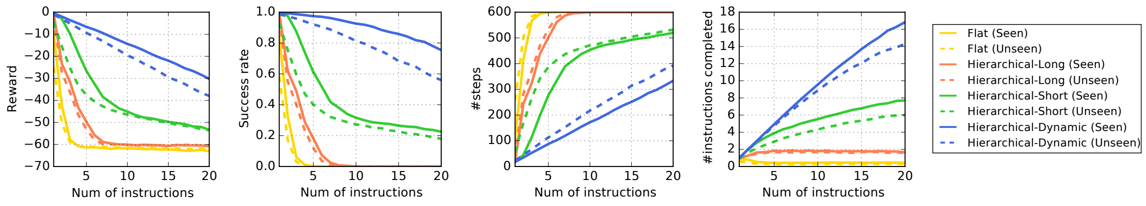


Figure 6.6: Performance per number of instructions. From left to right, the plots show reward, success rate, the number of steps, and the average number of instructions completed respectively.

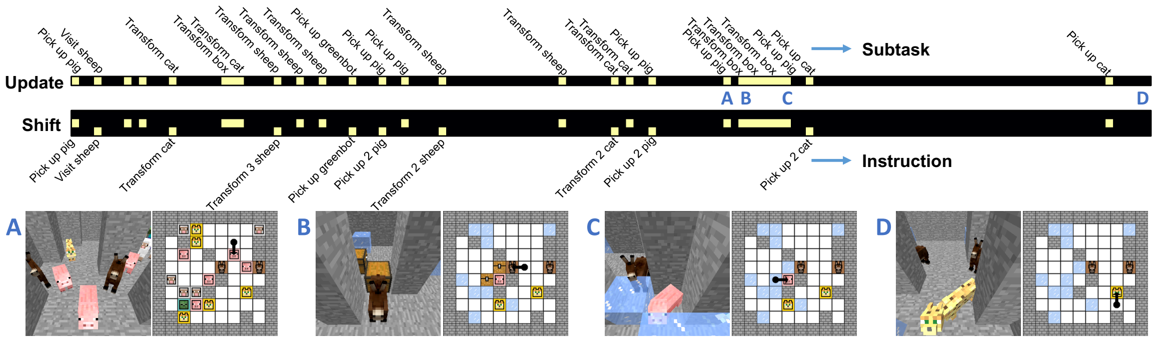


Figure 6.7: Analysis of the learned policy. 'Update' shows our agent's internal update decision. 'Shift' shows our agent's instruction-shift decision (-1, 0, and +1 from top to bottom). The bottom text shows the instruction indicated by the instruction pointer, while the top text shows the subtask chosen by the meta controller. (A) the agent picks up the pig to finish the instruction and moves to the next instruction. (B) When the agent observes a box that randomly appeared while executing 'Pick up 2 pig' instruction, it immediately changes its subtask to [Transform, Box]. (C) After dealing with the event (transforming a box), the agent resumes executing the instruction ('Pick up 2 pig'). (D) The agent finishes the final instruction.

- **Hierarchical-Long**: identical to our architecture except that the meta controller can update the subtask only when the current subtask is finished. This approach is similar to recent hierarchical deep RL methods (Kulkarni et al., 2016a; Tessler et al., 2017).
- **Hierarchical-Short**: identical to our architecture except that the meta controller updates the subtask at every time-step.

Overall Performance. The results on the instruction execution are summarized in Table 6.2 and Figure 6.6. It shows that our architecture ('**Hierarchical-Dynamic**') can handle a relatively long list of seen and unseen instructions of length 20 with reasonably high success rates, even though it is trained on short instructions of length 4. Although the performance degrades as the number of instructions increases, our architecture finishes 18 out of 20 seen instructions and 14 out of 20 unseen instructions on average. These results show that our agent is able to generalize to longer compositions of seen/unseen instructions by just learning to solve short sequences of a subset of instructions.

Flat vs. Hierarchy. Table 6.2 shows that the flat baseline completely fails even on training instructions. The flat controller tends to struggle with loop instructions (e.g., Pick up 3 pig) so that it learned a sub-optimal policy which moves to the next instruction with a small probability at each step regardless of its progress. This implies that it is hard for the flat controller to detect precisely when a subtask is finished, whereas hierarchical architectures can easily detect when a subtask is done, because the parameterized skill provides a termination signal to the meta controller.

Effect of Learned Time-Scale. As shown in Table 6.2 and Figure 6.6, 'Hierarchical-Long' baseline performs significantly worse than our architecture. We found that whenever a subtask is finished, this baseline puts a high probability to switch to [Transform, Box] regardless of the existence of box because transforming a box gives a bonus reward if a box exists by chance. However, this leads to wasting too much time finding a box until it appears and results in a poor success rate due to the time limit. This result implies that an open-loop policy that has to wait until a subtask finishes can be confused by such an uncertain event because it cannot interrupt ongoing subtasks before termination.

On the other hand, we observed that 'Hierarchical-Short' often fails on loop instructions by moving on to the next instruction before it finishes such instructions. This baseline should repeat the same subtask while not changing the instruction pointer for a long time and the reward is even more delayed given loop instructions. In contrast, the subtask updater in our architecture makes fewer decisions by operating at a large time-scale so that it can get more direct feedback from the long-term future. We conjecture that this is why our architecture performs better than this baseline. This result shows that learning when to update the subtask using the neural network is beneficial for dealing with delayed reward without compromising the ability to interrupt.

Analysis of The Learned Policy. We visualized our agent's behavior given a long list of instructions in Figure 6.7. Interestingly, when the agent sees a box, the meta controller immediately changes its subtask to [Transform, Box] to get a positive reward even though its instruction pointer is indicating 'Pick up 2 pig' and resumes executing the instruction after dealing with the box. Throughout this event and the loop instruction, the meta controller keeps the instruction pointer unchanged as illustrated in (B-C) in Figure 6.7. In addition, the agent learned to update the instruction pointer and the subtask almost only when it is needed, which provides the subtask updater with temporally-extended actions. This is not only computationally efficient but also useful for learning a better policy.

6.5 Discussion

In this chapter, we explored a type of zero-shot task generalization in RL with a new problem where the agent is required to execute and generalize over sequences of instructions. We proposed an analogy-making objective which enables generalization over unseen parameterized tasks in various scenarios. We also proposed a novel way to learn the time-scale of the meta controller that proved to be more efficient and flexible than alternative approaches for interrupting subtasks and for dealing with delayed sequential decision problems. Our empirical results on a stochastic 3D domain showed that our architecture generalizes well to longer sequences of instructions as well as unseen instructions. Although our hierarchical RL architecture was demonstrated in the simple setting where the set of instructions should be executed sequentially, we believe that our key ideas are not limited to this setting but can be extended to richer forms of instructions.

CHAPTER VII

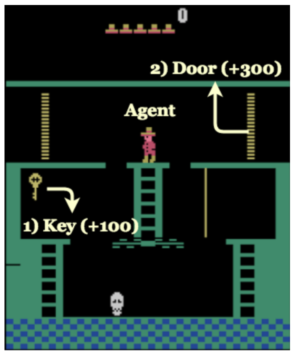
Self-Imitation Learning

This chapter proposes *Self-Imitation Learning* (SIL), a simple off-policy actor-critic algorithm that learns to reproduce the agent's past *good* decisions. This algorithm is designed to verify our hypothesis that exploiting past good experiences can indirectly drive deep exploration. Our empirical results show that SIL significantly improves advantage actor-critic (A2C) on several hard exploration Atari games and is competitive to the state-of-the-art count-based exploration methods. We also show that SIL improves proximal policy optimization (PPO) on MuJoCo tasks.

7.1 Introduction

The trade-off between exploration and exploitation is one of the fundamental challenges in reinforcement learning (RL). The agent needs to *exploit* what it already knows in order to maximize reward. But, the agent also needs to *explore* new behaviors in order to find a potentially better policy. The resulting performance of an RL agent emerges from this interaction between exploration and exploitation.

This chapter studies how exploiting the agent's past experiences improves learning in RL. More specifically, we hypothesize that learning to reproduce past good experiences can indirectly lead to deeper exploration depending on the domain. A simple example of how this can occur can be seen through our results on an example Atari game, Montezuma's Revenge (see Figure 7.1). In this domain, the first and more proximal source of reward is obtained by picking up the key. Obtaining the key is a precondition of the second and more distal source of reward (i.e., opening the door with the key). Many existing methods occasionally generate experiences that pick up the key and obtain the first reward, but fail to exploit these experiences often enough to learn how to open the door by exploring after picking up the key. Thus, they end up with a poor policy (see A2C in Figure 7.1). On the



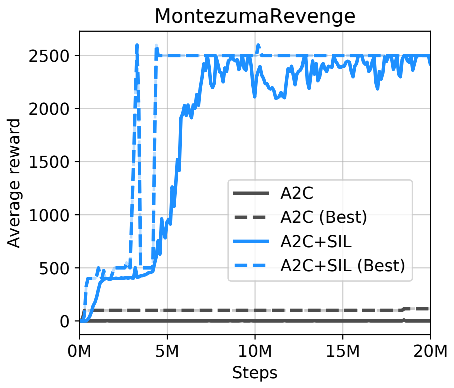


Figure 7.1: Learning curves on Montezuma's Revenge. (Left) The agent needs to pick up the key in order to open the door. Picking up the key gives a small reward. (Right) The baseline (A2C) often picks up the key as shown by the best episode reward in recent 100K steps (A2C (Best)), but it fails to consistently reproduce such a trajectory. In contrast, self-imitation learning (A2C+SIL) quickly learns to pick up the key as soon as the agent experiences it, which leads to the next source of reward (door).

other hand, by exploiting the experiences that pick up the key, the agent is able to explore onwards from the state where it has the key to successfully learn how to open the door (see A2C+SIL in Figure 7.1). Of course, this sort of exploitation can also hurt performance in problems where there are proximal distractor rewards and repeated exploitation of such rewards does not help in learning about more distal and higher rewards; in other words, these two aspects may both be present. In this chapter we will empirically investigate many different domains to see how exploiting past experiences can be beneficial for learning agents.

The main contributions of this chapter are as follows: (1) To study how exploiting past good experiences affects learning, we propose a *Self-Imitation Learning* (SIL) algorithm which learns to imitate the agent's own past good decisions. In brief, the SIL algorithm stores experiences in a replay buffer, learns to imitate state-action pairs in the replay buffer only when the return in the past episode is greater than the agent's value estimate. (2) We provide a theoretical justification of the SIL objective by showing that the SIL objective is derived from the lower bound of the optimal Q-function. (3) The SIL algorithm is very simple to implement and can be applied to any actor-critic architecture. (4) We demonstrate that SIL combined with advantage actor-critic (A2C) is competitive to the state-of-the-art count-based exploration actor-critic methods (e.g., Reactor-PixelCNN (Ostrovski et al., 2017)) on several hard exploration Atari games (Bellemare et al., 2013); SIL also improves the overall performance of A2C across 49 Atari games. Finally, SIL improves the performance of proximal policy optimization (PPO) on MuJoCo continuous control tasks (Brockman et al., 2016; Todorov et al., 2012), demonstrating that SIL may be generally applicable to any actor-critic architecture.

7.2 Related Work

Exploration There has been a long history of work on improving exploration in RL, including recent work that can scale up to large state spaces Stadie et al. (2015); Osband et al. (2016); Bellemare et al. (2016); Ostrovski et al. (2017). Many existing methods use some notion of curiosity or uncertainty as a signal for exploration (Schmidhuber, 1991; Strehl and Littman, 2008). In contrast, this chapter focuses on exploiting past good experiences for better exploration. Though the role of exploitation for exploration has been discussed (Thrun, 1992), prior work has mostly considered exploiting what the agent has learned, whereas we consider exploiting what the agent has experienced, but has not yet learned.

Episodic control Episodic control (Lengyel and Dayan, 2008) can be viewed as an extreme way of exploiting past experiences in the sense that the agent repeats the same actions that gave the best outcome in the past. MFEC (Blundell et al., 2016) and NEC (Pritzel et al., 2017) scaled up this idea to complex domains. However, these methods are slow during test-time because the agent needs to retrieve relevant states for each step and may generalize poorly as the resulting policy is non-parametric.

Experience replay Experience replay (Lin, 1992) is a natural way of exploiting past experiences for parametric policies. Prioritized experience replay (Moore and Atkeson, 1992; Schaul et al., 2016) proposed an efficient way of learning from past experiences by prioritizing them based on temporal-difference error. Our self-imitation learning also prioritizes experiences based on the full episode rewards. Optimality tightening (He et al., 2017) introduced an objective based on the lower/upper bound of the optimal Q-function, which is similar to a part of our theoretical result. These recent advances in experience replay have focused on value-based methods such as Q-learning, and are not easily applicable to actor-critic architectures.

Experience replay for actor-critic In fact, actor-critic framework (Sutton et al., 1999a; Konda and Tsitsiklis, 2000) can also utilize experience replay. Many existing methods are based on off-policy policy evaluation (Precup et al., 2001, 2000), which involves importance sampling. For example, ACER (Wang et al., 2017) and Reactor (Gruslys et al., 2018) use Retrace (Munos et al., 2016) to evaluate the learner from the behavior policy. Due to importance sampling, this approach may not benefit much from the past experience if the policy in the past is very different from the current policy. Although DPG (Silver et al., 2014; Lillicrap et al., 2016) performs experience replay without importance sampling, it is limited to continuous control. Our self-imitation learning objective does not involve importance sampling and is applicable to both discrete and continuous control.

Connection between policy gradient and Q-learning The recent studies on the relationship between policy gradient and Q-learning have shown that entropy-regularized policy gradient and Q-learning are closely related or even equivalent depending on assumptions (Nachum et al., 2017; O'Donoghue et al., 2017; Schulman et al., 2017a; Haarnoja et al., 2017). Our application of self-imitation learning to actor-critic (A2C+SIL) can be viewed as an instance of PGQL (O'Donoghue et al., 2017) in that we perform Q-learning on top of actor-critic architecture (see Section 7.4). Unlike Q-learning in PGQL, however, we use the proposed lower bound Q-learning to exploit good experiences.

Learning from imperfect demonstrations A few studies have attempted to learn from imperfect demonstrations, such as DQfD (Hester et al., 2018), Q-filter (Nair et al., 2017), and normalized actor-critic (Xu et al., 2018). Our self-imitation learning has a similar flavor in that the agent learns from imperfect demonstrations. However, we treat the agent's own experiences as demonstrations without using expert demonstrations. Although a similar idea has been discussed for program synthesis (Liang et al., 2016; Abolafia et al., 2018), this prior work used classification loss without justification. On the other hand, we propose a new objective, provide a theoretical justification, and systematically investigate how it drives exploration in RL.

7.3 Self-Imitation Learning

The goal of self-imitation learning (SIL) is to imitate the agent's past good experiences in the actor-critic framework. To this end, we propose to store past episodes with cumulative rewards in a replay buffer: $\mathcal{D} = \{(s_t, a_t, R_t)\}$, where s_t, a_t are a state and an action at time-step t, and $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$ is the discounted sum of rewards with a discount factor γ. To exploit only good state-action pairs in the replay buffer, we propose the following

Algorithm 6 Actor-Critic with Self-Imitation Learning

```
Initialize parameter $\theta$
Initialize replay buffer $\mathcal{D} \leftarrow \emptyset$
Initialize episode buffer $\mathcal{E} \leftarrow \emptyset$
for each iteration do
    # Collect on-policy samples
    for each step do
        Execute an action $s_t, a_t, r_t, s_{t+1} \sim \pi_\theta(a_t|s_t)$
        Store transition $\mathcal{E} \leftarrow \mathcal{E} \cup \{(s_t, a_t, r_t)\}$
    end for
    if $s_{t+1}$ is terminal then
        # Update replay buffer
        Compute returns $R_t = \sum_k^\infty \gamma^{k-t} r_k$ for all $t$ in $\mathcal{E}$
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, R_t)\}$ for all $t$ in $\mathcal{E}$
        Clear episode buffer $\mathcal{E} \leftarrow \emptyset$
    end if
    # Perform actor-critic using on-policy samples
    $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}^{a2c}$                                   (Eq. 7.4)
    # Perform self-imitation learning
    for $m = 1$ to $M$ do
        Sample a mini-batch $\{(s, a, R)\}$ from $\mathcal{D}$
        $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}^{sil}$                               (Eq. 7.1)
    end for
end for
```

off-policy actor-critic loss:

$$\mathcal{L}^{sil} = \mathbb{E}_{s,a,R\in\mathcal{D}}\left[\mathcal{L}^{sil}_{policy} + \beta^{sil}\mathcal{L}^{sil}_{value}\right] \tag{7.1}$$

$$\mathcal{L}^{sil}_{policy} = -\log \pi_\theta(a|s)\left(R - V_\theta(s)\right)_+ \tag{7.2}$$

$$\mathcal{L}^{sil}_{value} = \frac{1}{2}\left\|(R - V_\theta(s))_+\right\|^2 \tag{7.3}$$

where $(\cdot)_+ = \max(\cdot, 0)$, and $\pi_\theta, V_\theta(s)$ are the policy (i.e., actor) and the value function parameterized by θ. $\beta^{sil} \in \mathbb{R}^+$ is a hyperparameter for the value loss.

Note that $\mathcal{L}^{sil}_{policy}$ can be viewed as policy gradient using the value $V_\theta(s)$ as the state-dependent baseline except that we use the off-policy Monte-Carlo return (R) instead of on-policy return. $\mathcal{L}^{sil}_{policy}$ can also be interpreted as cross entropy loss (i.e., classification loss for discrete action) with sample weights proportional to the gap between the return and the agent's value estimate ($R - V_\theta$). If the return in the past is greater than the agent's value estimate ($R > V_\theta$), the agent learns to choose the action chosen in the past in the given state. Otherwise ($R \leq V_\theta$), and such a state-action pair is not used to update the parameter due to the $(\cdot)_+$ operator. This encourages the agent to imitate its own decisions in the past only when such decisions resulted in larger returns than expected. $\mathcal{L}^{sil}_{value}$ updates the value estimate towards the off-policy return R.

Prioritized Replay The proposed self-imitation learning objective $\mathcal{L}^{sil}$ is based on our theoretical result discussed in Section 7.4. In theory, the replay buffer ($\mathcal{D}$) can be any trajectories from any policies. However, only *good* state-action pairs that satisfy $R > V_\theta$ can contribute to the gradient during self-imitation learning (Eq. 7.1). Therefore, in order to get many state-action pairs that satisfy $R > V_\theta$, we propose to use the prioritized experience replay (Schaul et al., 2016). More specifically, we sample transitions from the replay buffer using the clipped advantage $(R - V_\theta(s))_+$ as priority (i.e., sampling probability is proportional to $(R - V_\theta(s))_+$). This naturally increases the proportion of *valid* samples that satisfy the constraint $(R - V_\theta(s))_+$ in SIL objective and thus contribute to the gradient.

Advantage Actor-Critic with SIL (A2C+SIL) Our self-imitation learning can be combined with any actor-critic method. In this chapter, we focus on the combination of advantage actor-critic (A2C) (Mnih et al., 2016) and self-imitation learning (A2C+SIL), as described in Algorithm 6. The objective of A2C ($\mathcal{L}^{a2c}$) is given by (Mnih et al., 2016):

$$\mathcal{L}^{a2c} = \mathbb{E}_{s,a\sim\pi_\theta}\left[\mathcal{L}^{a2c}_{policy} + \beta^{a2c}\mathcal{L}^{a2c}_{value}\right] \tag{7.4}$$

$$\mathcal{L}^{a2c}_{policy} = -\log\pi_\theta(a_t|s_t)(V^n_t - V_\theta(s_t)) - \alpha\mathcal{H}^{\pi_\theta}_t \tag{7.5}$$

$$\mathcal{L}^{a2c}_{value} = \frac{1}{2}\left\|V_\theta(s_t) - V^n_t\right\|^2 \tag{7.6}$$

where $\mathcal{H}^\pi_t = -\sum_a \pi(a|s_t)\log\pi(a|s_t)$ denotes the entropy in simplified notation, and α is a weight for entropy regularization. $V^n_t = \sum_{d=0}^{n-1}\gamma^d r_{t+d} + \gamma^n V_\theta(s_{t+n})$ is the n-step bootstrapped value.

To sum up, A2C+SIL performs both on-policy A2C update ($\mathcal{L}^{a2c}$) and self-imitation learning from the replay buffer M times ($\mathcal{L}^{sil}$) to exploit past good experiences. A2C+SIL is relatively simple to implement as it does not involve importance sampling.

7.4 Theoretical Justification

In this section, we justify the following claim.

Claim VII.1. *The self-imitation learning objective ($\mathcal{L}^{sil}$ in Eq. 7.1) can be viewed as an implementation of lower-bound-soft-Q-learning (Section 7.4.2) under the entropy-regularized RL framework.*

To show the above claim, we first introduce the entropy-regularized RL (Haarnoja et al., 2017) in Section 7.4.1. Section 7.4.2 introduces *lower-bound-soft-Q-learning*, an off-policy Q-learning algorithm, which learns the optimal action-value function from good state-action pairs. Section 7.4.3 proves the above claim by showing the equivalence between self-imitation learning and lower-bound-soft-Q-learning. Section 7.4.4 further discusses the relationship between A2C and self-imitation learning.

7.4.1 Entropy-Regularized Reinforcement Learning

The goal of entropy-regularized RL is to learn a stochastic policy which maximizes the entropy of the policy as well as the γ-discounted sum of rewards (Haarnoja et al., 2017; Ziebart et al., 2008):

$$\pi^* = \text{argmax}_\pi \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \left(r_t + \alpha \mathcal{H}_t^\pi \right) \right] \tag{7.7}$$

where $\mathcal{H}_t^\pi = -\log \pi(a_t|s_t)$ is the entropy of the policy π, and $\alpha \geq 0$ represents the weight of entropy bonus. Intuitively, in the entropy-regularized RL, a policy that has a high entropy is preferred (i.e., diverse actions chosen given the same state).

The optimal *soft* Q-function and the optimal *soft* value function are defined as:

$$Q^*(s_t, a_t) = \mathbb{E}_{\pi^*} \left[r_t + \sum_{k=t+1}^{\infty} \gamma^{k-t} (r_k + \alpha \mathcal{H}_k^{\pi^*}) \right] \tag{7.8}$$

$$V^*(s_t) = \alpha \log \sum_a \exp\left(Q^*(s_t, a)/\alpha \right). \tag{7.9}$$

It is shown that the optimal policy π^* has the following form (see Ziebart (2010); Haarnoja et al. (2017) for the proof):

$$\pi^*(a|s) = \exp((Q^*(s,a) - V^*(s))/\alpha). \tag{7.10}$$

This result provides the relationship among the optimal Q-function, the optimal policy, and the optimal value function, which will be useful in Section 7.4.3.

7.4.2 Lower Bound Soft Q-Learning

Lower bound of optimal soft Q-value Let π^* be an optimal policy in entropy-regularized RL (Eq. 7.7). It is straightforward that the expected return of any behavior policy μ can serve as a lower bound of the optimal soft Q-value as follows:

$$Q^*(s_t, a_t) = \mathbb{E}_{\pi^*}\left[r_t + \sum_{k=t+1}^{\infty} \gamma^{k-t}(r_k + \alpha \mathcal{H}_k^{\pi^*})\right] \tag{7.11}$$

$$\geq \mathbb{E}_{\mu}\left[r_t + \sum_{k=t+1}^{\infty} \gamma^{k-t}(r_k + \alpha \mathcal{H}_k^{\mu})\right], \tag{7.12}$$

because the entropy-regularized return of the optimal policy is always greater or equal to that of any other policies.

Lower bound soft Q-learning Suppose that we have full episode trajectories from a behavior policy μ, which consists of state-action-return triples: (s_t, a_t, R_t) where $R_t = r_t + \sum_{k=t+1}^{\infty} \gamma^{k-t}(r_k + \alpha \mathcal{H}_k^{\mu})$ is the entropy-regularized return. We propose *lower bound soft Q-learning* which updates $Q_\theta(s, a)$ parameterized by θ towards the optimal soft Q-value as follows (t is omitted for brevity):

$$\mathcal{L}^{lb} = \mathbb{E}_{s,a,R\sim\mu}\left[\frac{1}{2}\left\|(R - Q_\theta(s, a))_+\right\|^2\right], \tag{7.13}$$

where $(\cdot)_+ = \max(\cdot, 0)$. Intuitively, we update the Q-value only when the return is greater than the Q-value estimate ($R > Q_\theta(s, a)$). This is justified by the fact that the lower bound (Eq. 7.12) implies that the estimated Q-value is lower than the optimal soft Q-value: $Q^*(s, a) \geq R > Q_\theta(s, a)$ when the environment is deterministic. Otherwise ($R \leq Q_\theta(s, a)$), such state-action pairs do not provide any useful information about the optimal soft Q-value, so they are not used for training. We call this lower-bound-soft-Q-learning as it updates Q-values towards the lower bounds of the optimal Q-values observed from the behavior policy.

7.4.3 Connection between SIL and Lower Bound Soft Q-Learning

In this section, we derive an equivalent form of lower-bound-soft-Q-learning (Eq. 7.13) for the actor-critic architecture and show a connection to self-imitation learning objective.

Suppose that we have a parameterized soft Q-function Q_θ. According to the form of optimal soft value function and optimal policy in the entropy-regularized RL (Eq. 7.9-7.10), it is natural to consider the following forms of a value function V_θ and a policy π_θ:

$$V_\theta(s) = \alpha \log \sum_a \exp(Q_\theta(s,a)/\alpha) \tag{7.14}$$

$$\pi_\theta(a|s) = \exp((Q_\theta(s,a) - V_\theta(s))/\alpha). \tag{7.15}$$

From these definitions, Q_θ can be written as:

$$Q_\theta(s,a) = V_\theta(s) + \alpha \log \pi_\theta(a|s). \tag{7.16}$$

For convenience, let us define the following:

$$\hat{R} = R - \alpha \log \pi_\theta(a|s) \tag{7.17}$$

$$\Delta = R - Q_\theta(s,a) = \hat{R} - V_\theta(s). \tag{7.18}$$

By plugging Eq. 7.16 into Eq. 7.13, we can derive the gradient estimator of lower-bound-soft-Q-learning for the actor-critic architecture as follows:

$$\begin{aligned}
&\nabla_\theta \mathbb{E}_{s,a,R\sim\mu}\left[\frac{1}{2}\left\|(R - Q_\theta(s,a))_+\right\|^2\right] && (7.19)\\
&= \mathbb{E}\left[-\nabla_\theta Q_\theta(s,a)\Delta_+\right] && (7.20)\\
&= \mathbb{E}\left[-\nabla_\theta\left(\alpha \log \pi_\theta(a|s) + V_\theta(s)\right)\Delta_+\right] && (7.21)\\
&= \mathbb{E}\left[-\alpha\nabla_\theta \log \pi_\theta(a|s)\Delta_+ - \nabla_\theta V_\theta(s)\Delta_+\right] && (7.22)\\
&= \mathbb{E}\left[\alpha\nabla_\theta \mathcal{L}^{lb}_{policy} - \nabla_\theta V_\theta(s)\Delta_+\right] && (7.23)\\
&= \mathbb{E}\left[\alpha\nabla_\theta \mathcal{L}^{lb}_{policy} - \nabla_\theta V_\theta(s)(R - Q_\theta(s,a))_+\right] && (7.24)\\
&= \mathbb{E}\left[\alpha\nabla_\theta \mathcal{L}^{lb}_{policy} - \nabla_\theta V_\theta(s)(\hat{R} - V_\theta(s))_+\right] && (7.25)\\
&= \mathbb{E}\left[\alpha\nabla_\theta \mathcal{L}^{lb}_{policy} + \nabla_\theta \frac{1}{2}\left\|(\hat{R} - V_\theta(s))_+\right\|^2\right] && (7.26)\\
&= \mathbb{E}\left[\alpha\nabla_\theta \mathcal{L}^{lb}_{policy} + \nabla_\theta \mathcal{L}^{lb}_{value}\right]. && (7.27)
\end{aligned}$$

Each loss term in Eq. 7.27 is given by:

$$\mathcal{L}^{lb}_{policy} = -\log \pi_\theta(a|s)\left(\hat{R} - V_\theta(s)\right)_+ \tag{7.28}$$

$$\mathcal{L}^{lb}_{value} = \frac{1}{2}\left\|(\hat{R} - V_\theta(s))_+\right\|^2. \tag{7.29}$$

Thus, $\mathcal{L}^{lb}_{policy} = \mathcal{L}^{sil}_{policy}$ and $\mathcal{L}^{lb}_{value} = \mathcal{L}^{sil}_{value}$ as $\alpha \to 0$ (see Eq. 7.2-7.3). This shows that the proposed self-imitation learning objective $\mathcal{L}^{sil}$ (Eq. 7.1) can be viewed as a form of lower-bound-soft-Q-learning (Eq. 7.13), but without explicitly optimizing for entropy bonus reward as $\alpha \to 0$. Since the lower-bound-soft-Q-learning directly updates the Q-value towards the lower bound of the optimal Q-value, self-imitation learning can be viewed as an algorithm that updates the policy (π_θ) and the value (V_θ) directly towards the optimal policy and the optimal value respectively.

7.4.4 Relationship between A2C and SIL

Intuitively, A2C updates the policy in the direction of increasing the expected return of the learner policy and enforces consistency between the value and the policy from on-policy trajectories. On the other hand, SIL updates each of them directly towards optimal policies and values respectively from off-policy trajectories. In fact, Nachum et al. (2017); Haarnoja et al. (2017); Schulman et al. (2017a) have recently shown that entropy-regularized A2C can be viewed as n-step online soft Q-learning (or path consistency learning). Therefore, both A2C and SIL objectives are designed to learn the optimal soft Q-function in the entropy-regularized RL framework. Thus, we claim that both objectives can be complementary to each other in that they share the same optimal solution as discussed in PGQL (O'Donoghue et al., 2017).

7.5 Experiments

The experiments are designed to answer the following:

- Is self-imitation learning useful for exploration?
- Is self-imitation learning complementary to count-based exploration methods?
- Does self-imitation learning improve the overall performance across a variety of tasks?
- When does self-imitation learning help and when does it not?
- Can other off-policy actor-critic methods also exploit good experiences (e.g., ACER (Wang et al., 2017))?
- Is self-imitation learning useful for continuous control and compatible with other learning algorithms like PPO (Schulman et al., 2017b)?

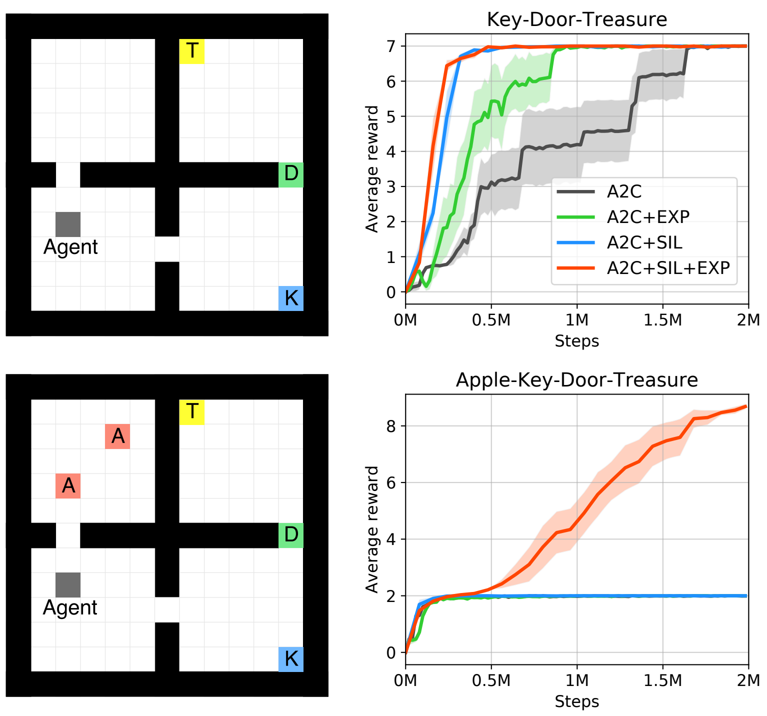


Figure 7.2: Key-Door-Treasure domain. The agent should pick up the key (K) in order to open the door (D) and collect the treasure (T) to maximize the reward. In the Apple-Key-Door-Treasure domain (bottom), there are two apples (A) that give small rewards (+1). ‘SIL’ and ‘EXP’ represent our self-imitation learning and a count-based exploration method respectively.

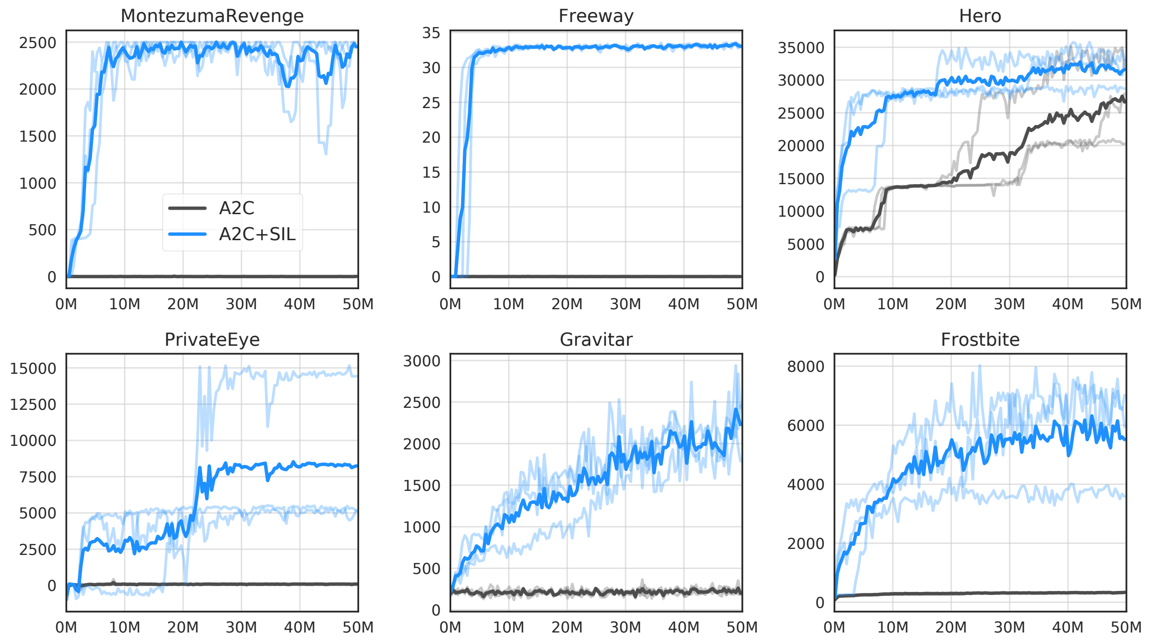


Figure 7.3: Learning curves on hard exploration Atari games. X-axis and y-axis represent steps and average reward respectively.

7.5.1 Implementation Details

For Atari experiments, we used a 3-layer convolutional neural network used in DQN (Mnih et al., 2015) with last 4 stacked frames as input. We performed 4 self-imitation learning updates per on-policy actor-critic update ($M = 4$ in Algorithm 6). Instead of treating losing a life as episode termination as typically done in the previous work, we terminated episodes when the game ends, as it is the true definition of episode termination. For MuJoCo experiments, we used an MLP which consists of 2 hidden layers with 64 units as in Schulman et al. (2017b). We performed 10 self-imitation learning updates per each iteration (batch). Our implementation is based on OpenAI's baseline implementation (Dhariwal et al., 2017).[1]

7.5.2 Key-Door-Treasure Domain

To investigate how self-imitation learning is useful for exploration and whether it is complementary to count-based exploration method, we compared different methods on a grid-world domain, as illustrated in Figure 7.2. More specifically, we implemented a count-based exploration method (Strehl and Littman, 2008) that gives an exploration bonus reward: $r_{exp} = \beta / \sqrt{N(s)}$, where $N(s)$ is the visit count of state s and β is a hyperparameter. We also implemented a combination with self-imitation learning shown as 'A2C+SIL+EXP' in Figure 7.2.

In the first domain (Key-Door-Treasure), the chance of picking up the key followed by opening the door and obtaining the treasure is low due to the sequential dependency between them. We found that the baseline A2C tends to get stuck at a sub-optimal policy that only opens the door for a long time. A2C+EXP learns faster than A2C because exploration bonus encourages the agent to collect the treasure more often. Interestingly, A2C+SIL and A2C+SIL+EXP learn most quickly. We observed that once the agent opens the door with the key by chance, our SIL helps exploit such good experiences and quickly learns to open the door with the key. This increases the chance of getting the next reward (i.e., treasure) and helps learn the optimal policy. This is an example showing that self-imitation learning can drive deep exploration.

In the second domain (Apple-Key-Door-Treasure), collecting apples near the agent's initial location makes it even more challenging for the agent to learn the optimal policy, which collects all of the objects within the time limit (50 steps). In this domain, many agents learned a sub-optimal policy that only collects two apples as shown in Figure 7.2. On the other hand, only A2C+SIL+EXP consistently learned the optimal policy because count-based exploration increases the chance of collecting the treasure, while self-imitation learning can

[1]The code is available on `https://github.com/junhyukoh/self-imitation-learning`.

Table 7.1: Comparison to count-based exploration actor-critic agents on hard exploration Atari games. A3C+ and Reactor+ correspond to A3C-CTS (Bellemare et al., 2016) and Reactor-PixelCNN respectively (Ostrovski et al., 2017). SimHash represents TRPO-AE-SimHash (Tang et al., 2017). †Numbers are taken from plots.

	A2C+SIL	A3C+	Reactor+†	SimHash
Montezuma	**2500**	273	100	75
Freeway	**34**	30	32	33
Hero	**33069**	15210	28000	N/A
PrivateEye	**8684**	99	200	N/A
Gravitar	**2722**	239	1600	482
Frostbite	**6439**	352	4800	5214
Venture	0	0	**1400**	445

quickly exploit such a good experience as soon as the agent collects it. This result shows that self-imitation learning and count-based exploration methods can be complementary to each other. This result also suggests that while exploration is important for increasing the chance/frequency of getting a reward, it is also important to exploit such rare experiences to learn a policy to consistently achieve it especially when the reward is sparse.

7.5.3 Hard Exploration Atari Games

We investigated how useful our self-imitation learning is for several hard exploration Atari games on which recent advanced exploration methods mainly focused. Figure 7.3 shows that A2C with our self-imitation learning (A2C+SIL) outperforms A2C on six hard exploration games. A2C failed to learn a better-than-random policy, except for Hero, whereas our method learned better policies and achieved human-level performances on Hero and Freeway. We observed that even a random exploration occasionally leads to a positive reward on these games, and self-imitation learning helps exploit such an experience to learn a good policy from it. This can drive deep exploration when the improved policy gets closer to the next source of reward. This result supports our claim that exploiting past experiences can often help exploration.

We further compared our method against the state-of-the-art count-based exploration actor-critic agents (A3C-CTS (Bellemare et al., 2016), Reactor-PixelCNN (Ostrovski et al., 2017), and SimHash (Tang et al., 2017)). These methods learn a density model of the observation or a hash function and use it to compute pseudo visit count, which is used to compute an exploration bonus reward. Even though our method does not have an explicit

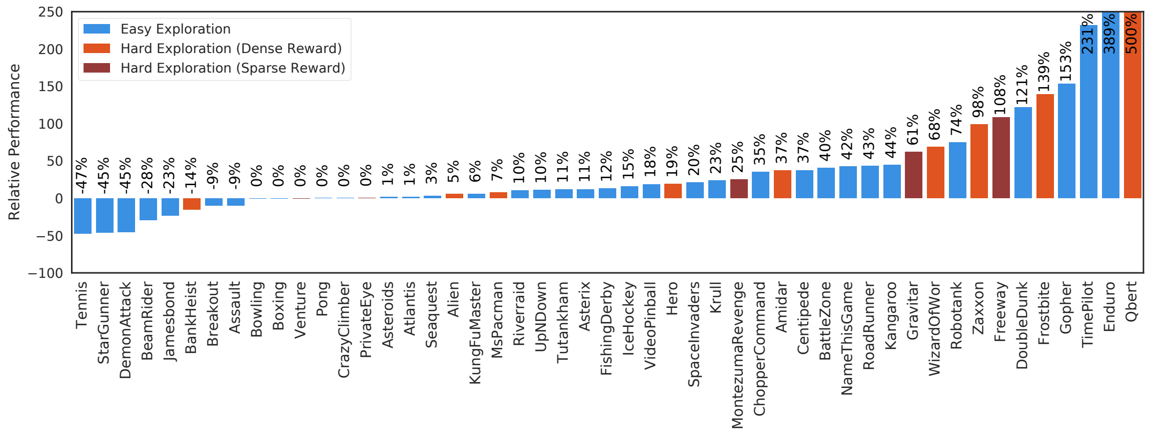


Figure 7.4: Relative performance of A2C+SIL over A2C across 49 Atari games.

exploration bonus that encourages exploration, we were curious how well our self-imitation learning approach performs compared to these exploration approaches.

Interestingly, Table 7.1 shows that A2C with our self-imitation learning (A2C+SIL) achieves better results on 6 out of 7 hard exploration games without any technique that explicitly encourages exploration. This result suggests that it is important to exploit past good experiences as well as efficiently explore the environment to drive deep exploration.

On the other hand, we found that A2C+SIL never receives a positive reward on Venture during training. This makes it impossible for our method to learn a good policy because there is no good experience to exploit, whereas one of the count-based exploration methods (Reactor-PixelCNN) achieves a better performance, because the agent is encouraged to explore different states even in the absence of reward signal from the environment. This result suggests that an advanced exploration method is essential in such environments where a random exploration never generates a good experience within a reasonable amount of time. Combining self-imitation learning with state-of-the-art exploration methods would be an interesting future work.

7.5.4 Overall Performance on Atari Games

To see how useful self-imitation learning is across various types of environments, we evaluated our self-imitation learning method on 49 Atari games. It turns out that our method (A2C+SIL) significantly outperforms A2C in terms of median human-normalized score as shown in Table 7.2. Figure 7.4 shows the relative performance of A2C+SIL compared to A2C using the measure proposed by Wang et al. (2016). It is shown that our method (A2C+SIL) improves A2C on 35 out of 49 games in total and 11 out of 14 hard exploration games defined by Bellemare et al. (2016). It is also shown that A2C+SIL performs significantly

Table 7.2: Performance of agents on 49 Atari games after 50M steps (200M frames) of training. 'ACPER' represents A2C with prioritized replay using ACER objective. 'Median' shows median of human-normalized scores. '>Human' shows the number of games where the agent outperforms human experts.

AGENT	MEDIAN	>HUMAN
A2C	96.1%	23
ACPER	46.8%	18
A2C+SIL	**138.7%**	**29**

better on many easy exploration games such as Time Pilot as well as hard exploration games. We observed that there is a certain learning stage where the agent suddenly achieves a high score by chance on such games, and our self-imitation learning exploits such experiences as soon as the agent experiences them.

On the other hand, we observed that our method often learns faster at the early stage of learning, but sometimes gets stuck at a sub-optimal policy on a few games, such as James Bond and Star Gunner. This suggests that excessive exploitation at the early stage of learning can hurt the performance. We found that reducing the number of SIL updates per iteration or using a small weight for the SIL objective in a later learning stage indeed resolves this issue and even improve the performance on such games, though the reported numbers are based on the single best hyperparameter. Thus, automatically controlling the degree of self-imitation learning would be an interesting future work.

7.5.5 Effect of Lower Bound Soft Q-Learning

A natural question is whether existing off-policy actor-critic methods can also benefit from past good experiences by exploiting them. To answer this question, we trained ACPER (A2C with prioritized experience replay) which performs off-policy actor-critic update proposed by ACER (Wang et al., 2017) by using the same prioritized experience replay as ours, which uses $(R - V_\theta)_+$ as sampling priority. ACPER can also be viewed as the original ACER with our proposed prioritized experience replay.

Table 7.2 shows that ACPER performs much worse than our A2C+SIL and is even worse than A2C. We observed that ACPER also benefits from good episodes on a few hard exploration games (e.g., Freeway) but was very unstable on many other games.

We conjecture that this is due to the fact that the ACER objective has an importance weight term $(\pi(a|s)/\mu(a|s))$. This approach may not benefit much from the good experiences in the past if the current policy deviates too much from the decisions made in the

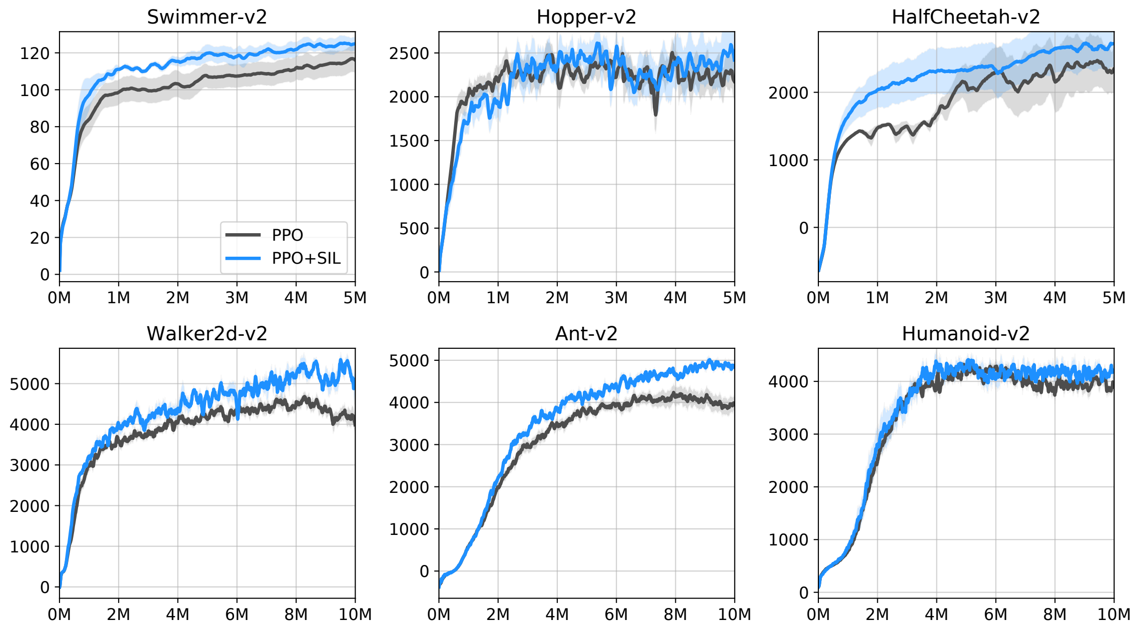


Figure 7.5: Performance on OpenAI Gym MuJoCo tasks. The learning curves are averaged over 10 random seeds.

past. On the other hand, the proposed self-imitation learning objective (Eq. 7.1) does not have an importance weight and can learn from any behavior, as long as the behavior policy performs better than the learner. This is because our gradient estimator can be interpreted as lower-bound-soft-Q-learning, which updates the parameter directly towards the optimal Q-value regardless of the similarity between the behavior policy and the learner as discussed in Section 7.4.2. This result shows that our self-imitation learning objective is suitable for exploiting past good experiences.

7.5.6 Performance on MuJoCo

This section investigates whether self-imitation learning is beneficial for continuous control tasks and whether it can be applied to other types of policy optimization algorithms, such as proximal policy optimization (PPO) (Schulman et al., 2017b). Note that unlike A2C, PPO does not have a strong theoretical connection to our SIL objective. However, we claim that they can still be complementary to each other in that both PPO and SIL try to update the policy and the value towards the optimal policy and value. To empirically verify this, we implemented PPO+SIL, which updates the parameter using both the PPO algorithm and our SIL algorithm and evaluated it on 6 MuJoCo tasks in OpenAI Gym (Brockman et al., 2016).

The result in Figure 7.5 shows that our self-imitation learning improves PPO on Swimmer, Walker2d, and Ant tasks. Unlike Atari games, the reward structure in this benchmark is

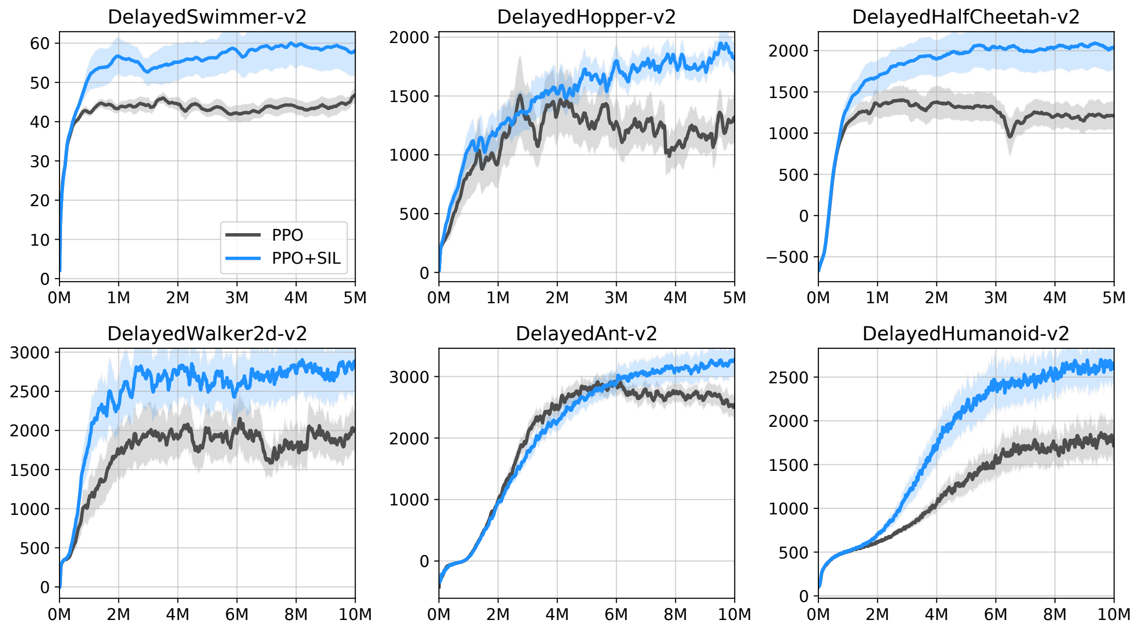


Figure 7.6: Performance on delayed-reward versions of OpenAI Gym MuJoCo tasks. The learning curves are averaged over 10 random seeds.

smooth and dense in that the agent always receives a reasonable amount of reward according to its continuous progress. We conjecture that the agent has a relatively low chance to occasionally perform well and learn much faster by exploiting such an experience in this type of domain. Nevertheless, the overall improvement suggests that self-imitation learning can be generally applicable to actor-critic architectures and a variety of tasks.

To verify our conjecture, we further conducted experiments by delaying reward the agent gets from the environment. More specifically, the modified tasks give an accumulated reward after every 20 steps (or when the episode terminates). This makes it more difficult to learn a good policy because the agent does not receive a reward signal for every step. The result is shown in the bottom row in Figure 7.5. Not surprisingly, we observed that both PPO and PPO+SIL perform worse on the delayed-reward tasks than themselves on the standard OpenAI Gym tasks. However, it is clearly shown that the gap between PPO+SIL and PPO is larger on delayed-reward tasks compared to standard tasks. Unlike the standard OpenAI Gym tasks where reward is well-designed and dense, we conjecture that the chance of achieving high overall rewards is much low in the delayed-reward tasks. Thus, the agent can benefit more from self-imitation learning because self-imitation learning captures such rare experiences and learn from them.

7.6 Discussion

In this chapter, we proposed self-imitation learning, which learns to reproduce the agent's past good experiences, and showed that self-imitation learning is very helpful on hard exploration tasks as well as a variety of other tasks including continuous control tasks. We also showed that a proper level of exploitation of past experiences during learning can drive deep exploration, and that self-imitation learning and exploration methods can be complementary. Our results suggest that there can be a certain learning stage where exploitation is more important than exploration or vice versa. Thus, we believe that developing methods for balancing between exploration and exploitation in terms of collecting and learning from experiences is an important future research direction.

CHAPTER VIII

Conclusion

8.1 Summary of Contributions

The goal of this thesis was to show how to learn a dynamics model of the environment and how to use it for lookahead planning, to show how deep neural networks can be used to develop the ability to generalize to unseen partially observable environments and unseen tasks, and to show how exploiting past good experiences improve sample efficiency by indirectly driving deep exploration.

In Chapter III, we introduced neural network architectures that can perform action-conditional predictions given high-dimensional visual observations. We showed that the proposed architectures can make reliable long-term predictions in Atari games. This chapter further investigated the usefulness of the learned model by using it for informed exploration.

In Chapter IV, we explored an alternative way to learn a model of the environment, called value prediction network (VPN), which learns to predict future rewards and values without predicting observations. VPN has both model-free and model-based RL components in a single neural network, which allows performing a lookahead tree search while estimating values at any step. We demonstrated that VPN is much more robust to the stochasticity of the environment compared to conventional model-based RL approaches.

In Chapter V, we introduced a set of cognitive tasks in a 3D partially observable environment using Minecraft, each of which requires the ability to remember important information in the past. We systematically evaluated different DRL architectures including our memory-based architectures in terms of generalization performance. The result showed that training performance does not guarantee generalization performance in partially observable environments, and memory-based architectures generalize better than existing architectures to unseen 3D environments.

In Chapter VI, we introduced a new multi-task RL problem where the agent is required to perform different tasks depending on given instructions and generalize to unseen instructions during evaluation. We proposed several objectives and hierarchical architectures that allow the agent to generalize well to unseen and longer sequential instructions just from a minimal knowledge about the test set of instructions.

In Chapter VII, we demonstrated our hypothesis that a proper level of exploitation of past good experiences can indirectly drive deep exploration in complex environments. To show this, we proposed self-imitation learning which learns from the agent's past good experiences. We demonstrated that self-imitation learning significantly improves the performance of an actor-critic agent across many different RL domains including hard exploration Atari games.

8.2 Future Directions and Open Problems

Model-based RL and Planning Despite recent advances in model-based RL and planning using DRL (Weber et al., 2017; Oh et al., 2017a), the benefit of lookahead planning is not significant compared to the state-of-the-art model-free RL agents on challenging RL benchmarks such as Atari games. However, most of the previous works have focused on learning and using a one-step forward model, which is not scalable due to the exponentially large search space and compounding error. On the other hand, humans can make long-term predictions in an abstract state space rather than raw observation space, which enables long-term planning. An interesting future direction is to explore ways to make long-term predictions in an abstract state space using temporal abstractions (i.e., temporally-extended actions (Precup, 2000)) and build an abstract planning module on top of it. Predictron (Silver et al., 2017b) showed a promising result in this direction, though it is limited to uncontrolled setting. This is clearly related to discovering temporal abstractions and hierarchical RL as discussed further below. In addition to planning, another interesting future direction in model-based RL is to use a learned model to generate *imaginary* samples for training the agent, such as Dyna architecture (Sutton et al., 2008) and Predictron (Silver et al., 2017b), to improve sample efficiency.

Hierarchy Discovering temporal abstractions (Precup, 2000; Sutton et al., 1999b) is one of the most important problems in RL. However, there has been no clear understanding of what objective drives the emergence of temporal abstractions especially in large-scale environments. Recently, Feudal Network (Vezhnevets et al., 2017) attempted to use the idea of separating rewards between 'manager' and 'worker' from Feudal architecture (Dayan and Hinton, 1993) to discover temporal abstractions. Another recent work by Frans et al. (2018) claimed that learning 'reusable' sub-policies that are useful across a distribution of tasks leads to the emergence of temporal abstractions. Though these ideas sound reasonable, all the previous works used a fixed time-scale for temporal abstractions (e.g., 10-100 steps). An open question is how to automatically discover both a sub-policy and corresponding temporal scale, which fully defines an option with its termination function (Sutton et al., 1999b). Discovering temporal abstractions seems also related to building hierarchical recurrent neural networks for sequence modeling (Koutnik et al., 2014; Chung et al., 2017) in the sense that the goal is to find a hierarchical solution for temporal credit assignment problems, which also remains an open problem. Developing an objective for discovering temporal hierarchy would be an important step for both RL and deep learning.

Exploration An efficient exploration is crucial for collecting useful and informative experiences in complex environments. The current state-of-the-art exploration approaches attempted to measure a notion of curisoity (Schmidhuber, 1991; Strehl and Littman, 2008) by measuring state visitation (Bellemare et al., 2016; Tang et al., 2017), information gain (Houthooft et al., 2016), and prediction error (Stadie et al., 2015; Pathak et al., 2017). A fundamental limitation of these approaches is that their exploration strategies are *passive* in the sense that the agent receives intrinsic rewards only after the agent discovers interesting states. On the other hand, humans are *proactive* in the sense that we become curious about new states even before we have ever visited them. For example, imagine a simple room with objects. After only a few steps in the environment, humans tend to become immediately interested in interaction with objects and try to reach them, even though they have never interacted with these objects before. This kind of proactive exploration behavior is not possible with the existing approaches, because there is no motivation for the agent to interact with such objects in the first place. Buildling such proactive exploration methods would be an interesting future direction as RL domains become increasingly complex and open-ended (Vinyals et al., 2017; Johnson et al., 2016).

Unsupervised Learning Unsupervised learning was the key to success in the early era of deep learning (LeCun et al., 2015; Lee, 2010; Hinton and Salakhutdinov, 2006). However, it remains an open question what are useful unsupervised learning signals for RL when there is no task of interest in the environment, or when the reward signal is extremely delayed and sparse. Using a form of intrinsic motivation (Singh et al., 2004) as an internal reward function seems necessary to encourage the agent to acquire knowledge about the environment in an unsupervised fashion. A natural form of intrinsic motivation is to learn to control many different aspects of the environment or to reach many different states in the environment (Kaelbling, 1993). There has been a line of recent work along this direction (Held et al., 2017; Sukhbaatar et al., 2018; Bengio et al., 2017; Jaderberg et al., 2017). Most of them use unsupervised learning signals as a way of pre-training the agent by initializing the weight of the neural networks from unsupervised learning. However, a more ambitious goal in RL would be to use unsupervised learning for *continual learning* (Ring, 1994) by continually distilling a set of skills (or temporal abstractions) from unsupervised learning and use it for further learning (e.g., further unsupervised learning or learning a main task when it comes), which would be one of the most essential components of AI in open-ended environments.

BIBLIOGRAPHY

BIBLIOGRAPHY

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Abolafia, D. A., Norouzi, M., and Le, Q. V. (2018). Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*.

Andre, D. and Russell, S. J. (2000). Programmable reinforcement learning agents. In *Advances in the Neural Information Processing System*.

Andre, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Andreas, J., Klein, D., and Levine, S. (2017). Learning modular neural network policies for multi-task and multi-robot transfer. In *Proceedings of the International Conference on Machine Learning*.

Bacon, P.-L., Harb, J., and Precup, D. (2017). The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations*.

Bakker, B., Zhumatiy, V., Gruener, G., and Schmidhuber, J. (2003). A robot that reinforcement-learns to identify and memorize important previous observations. In *Intelligent Robots and Systems*, volume 1, pages 430–435.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Advances in the Neural Information Processing System*.

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

Bellemare, M. G., 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. G., Veness, J., and Bowling, M. (2012). Investigating contingency awareness using Atari 2600 games. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

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

Bengio, E., Thomas, V., Pineau, J., Precup, D., and Bengio, Y. (2017). Independently controllable features. *arXiv preprint arXiv:1703.07718*.

Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the International Conference on Machine Learning*.

Blundell, C., Uria, B., Pritzel, A., Li, Y., Ruderman, A., Leibo, J. Z., Rae, J., Wierstra, D., and Hassabis, D. (2016). Model-free episodic control. *arXiv preprint arXiv:1606.04460*.

Branavan, S. R. K., Chen, H., Zettlemoyer, L. S., and Barzilay, R. (2009). Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43.

Bubic, A., Von Cramon, D. Y., and Schubotz, R. I. (2010). Prediction, cognition and the brain. *Frontiers in human neuroscience*, 4.

Chen, D. L. and Mooney, R. J. (2011). Learning to interpret natural language navigation instructions from observations. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Chiappa, S., Racaniere, S., Wierstra, D., and Mohamed, S. (2017). Recurrent environment simulators. In *Proceedings of the International Conference on Learning Representations*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Chung, J., Ahn, S., and Bengio, Y. (2017). Hierarchical multiscale recurrent neural networks. In *Proceedings of the International Conference on Learning Representations*.

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

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

da Silva, B. C., Konidaris, G., and Barto, A. G. (2012). Learning parameterized skills. In *Proceedings of the International Conference on Machine Learning*.

Dayan, P. and Hinton, G. E. (1993). Feudal reinforcement learning. In *Advances in the Neural Information Processing System*. Morgan Kaufmann Publishers.

Devin, C., Gupta, A., Darrell, T., Abbeel, P., and Levine, S. (2017). Learning modular neural network policies for multi-task and multi-robot transfer. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2017). Openai baselines. https://github.com/openai/baselines.

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

Dosovitskiy, A., Springenberg, J. T., and Brox, T. (2015). Learning to generate chairs with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Finn, C., Goodfellow, I. J., and Levine, S. (2016). Unsupervised learning for physical interaction through video prediction. In *Advances in the Neural Information Processing System*.

Finn, C. and Levine, S. (2017). Deep visual foresight for planning robot motion. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Florensa, C., Duan, Y., and Abbeel, P. (2017). Stochastic neural networks for hierarchical reinforcement learning. In *Proceedings of the International Conference on Learning Representations*.

Frans, K., Ho, J., Chen, X., Abbeel, P., and Schulman, J. (2018). Meta learning shared hierarchies. In *Proceedings of the International Conference on Learning Representations*.

Ghavamzadeh, M. and Mahadevan, S. (2003). Hierarchical policy gradient algorithms. In *Proceedings of the International Conference on Machine Learning*.

Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Gruslys, A., Azar, M. G., Bellemare, M. G., and Munos, R. (2018). The reactor: A sample-efficient actor-critic architecture. In *Proceedings of the International Conference on Learning Representations*.

Gu, S., Lillicrap, T. P., Sutskever, I., and Levine, S. (2016). Continuous deep q-learning with model-based acceleration. In *Proceedings of the International Conference on Machine Learning*.

Guo, X., Singh, S., Lee, H., Lewis, R. L., and Wang, X. (2014). Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in the Neural Information Processing System*.

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

Haarnoja, T., Tang, H., Abbeel, P., and Levine, S. (2017). Reinforcement learning with deep energy-based policies. In *Proceedings of the International Conference on Machine Learning*.

Hadsell, R., Chopra, S., and LeCun, Y. (2006). Dimensionality reduction by learning an invariant mapping. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Hausknecht, M. and Stone, P. (2015). Deep recurrent q-learning for partially observable MDPs. *arXiv preprint arXiv:1507.06527*.

He, F. S., Liu, Y., Schwing, A. G., and Peng, J. (2017). Learning to play in a day: Faster deep reinforcement learning by optimality tightening. In *Proceedings of the International Conference on Learning Representations*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Heess, N., Wayne, G., Silver, D., Lillicrap, T. P., Tassa, Y., and Erez, T. (2015). Learning continuous control policies by stochastic value gradients. In *Advances in the Neural Information Processing System*.

Heess, N., Wayne, G., Tassa, Y., Lillicrap, T. P., Riedmiller, M. A., and Silver, D. (2016). Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*.

Held, D., Geng, X., Florensa, C., and Abbeel, P. (2017). Automatic goal generation for reinforcement learning agents. *arXiv preprint arXiv:1705.06366*.

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Sendonaris, A., Dulac-Arnold, G., Osband, I., Agapiou, J., et al. (2018). Deep q-learning from demonstrations. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97.

Hinton, G. E. and Salakhutdinov, R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313 5786:504–7.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Houthooft, R., Chen, X., Duan, Y., Schulman, J., Turck, F. D., and Abbeel, P. (2016). Vime: Variational information maximizing exploration. In *Advances in the Neural Information Processing System*.

Isele, D., Rostami, M., and Eaton, E. (2016). Using task features for zero-shot knowledge transfer in lifelong learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Jaderberg, M., Mnih, V., Czarnecki, W., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2017). Reinforcement learning with unsupervised auxiliary tasks. In *Proceedings of the International Conference on Learning Representations*.

James, W. (2013). *The principles of psychology*. Read Books Ltd.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *ACM Multimedia*.

Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. (2016). The malmo platform for artificial intelligence experimentation. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Joulin, A. and Mikolov, T. (2015). Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. In *Advances in the Neural Information Processing System*.

Kaelbling, L. P. (1993). Learning to achieve goals. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Kalchbrenner, N., van den Oord, A., Simonyan, K., Danihelka, I., Vinyals, O., Graves, A., and Kavukcuoglu, K. (2016). Video pixel networks. *arXiv preprint arXiv:1610.00527*.

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

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

Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274.

Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *Machine Learning*, pages 282–293. Springer.

Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-critic algorithms. In *Advances in the Neural Information Processing System*.

Konidaris, G. and Barto, A. G. (2007). Building portable options: Skill transfer in reinforcement learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Konidaris, G., Scheidwasser, I., and Barto, A. G. (2012). Transfer in reinforcement learning via shared features. *Journal of Machine Learning Research*, 13:1333–1371.

Koutnik, J., Greff, K., Gomez, F., and Schmidhuber, J. (2014). A clockwork rnn. In *Proceedings of the International Conference on Machine Learning*.

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

Kulkarni, T. D., Narasimhan, K. R., Saeedi, A., and Tenenbaum, J. B. (2016a). Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in the Neural Information Processing System*.

Kulkarni, T. D., Saeedi, A., Gautam, S., and Gershman, S. (2016b). Deep successor reinforcement learning. *arXiv preprint arXiv:1606.02396*.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2016). Building machines that learn and think like people. *The Behavioral and brain sciences*, pages 1–101.

Lakshminarayanan, A. S., Sharma, S., and Ravindran, B. (2017). Dynamic action repetition for deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Lange, S. and Riedmiller, M. (2010). Deep auto-encoder neural networks in reinforcement learning. In *Proceedings of the International Joint Conference on Neural Networks*.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Lee, H. (2010). *Unsupervised feature learning via sparse hierarchical representations*. PhD thesis, Stanford University.

Lengyel, M. and Dayan, P. (2008). Hippocampal contributions to control: the third way. In *Advances in the Neural Information Processing System*.

Lenz, I., Knepper, R. A., and Saxena, A. (2015). DeepMPC: Learning deep latent features for model predictive control. In *Robotics: Science and Systems*.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(1):1334–1373.

Levine, S. and Koltun, V. (2013). Guided policy search. In *Proceedings of the International Conference on Machine Learning*.

Li, L., Chu, W., Langford, J., and Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the International Conference on World Wide Web*, pages 661–670. ACM.

Liang, C., Berant, J., Le, Q., Forbus, K. D., and Lao, N. (2016). Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of The Annual Meeting of the Association for Computational Linguistics*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. In *Proceedings of the International Conference on Learning Representations*.

Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321.

MacMahon, M., Stankiewicz, B., and Kuipers, B. (2006). Walk the talk: Connecting language, knowledge, and action in route instructions. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

McGovern, A. and Barto, A. G. (2002). *Autonomous discovery of temporal abstractions from interaction with an environment*. PhD thesis, University of Massachusetts.

Mei, H., Bansal, M., and Walter, M. R. (2016). Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Memisevic, R. (2013). Learning to relate images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1829–1846.

Michalski, V., Memisevic, R., and Konda, K. (2014). Modeling deep temporal dependencies with recurrent grammar cells. In *Advances in the Neural Information Processing System*.

Mikolov, T., Le, Q. V., and Sutskever, I. (2013). Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*.

Mishra, N., Abbeel, P., and Mordatch, I. (2017). Prediction and control with temporal segment models. In *Proceedings of the International Conference on Machine Learning*.

Mittelman, R., Kuipers, B., Savarese, S., and Lee, H. (2014). Structured recurrent temporal restricted Boltzmann machines. In *Proceedings of the International Conference on Machine Learning*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Moore, A. W. and Atkeson, C. G. (1992). Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In *Advances in the Neural Information Processing System*.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130.

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. G. (2016). Safe and efficient off-policy reinforcement learning. In *Advances in the Neural Information Processing System*.

Nachum, O., Norouzi, M., Xu, K., and Schuurmans, D. (2017). Bridging the gap between value and policy based reinforcement learning. In *Advances in the Neural Information Processing System*.

Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W., and Abbeel, P. (2017). Overcoming exploration in reinforcement learning with demonstrations. *arXiv preprint arXiv:1709.10089*.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning*.

Narasimhan, K., Kulkarni, T., and Barzilay, R. (2015). Language understanding for text-based games using deep reinforcement learning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

O'Donoghue, B., Munos, R., Kavukcuoglu, K., and Mnih, V. (2017). Combining policy gradient and q-learning. In *Proceedings of the International Conference on Learning Representations*.

Oh, J., Chockalingam, V., Singh, S., and Lee, H. (2016). Control of memory, active perception, and action in minecraft. In *Proceedings of the International Conference on Machine Learning*.

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

Oh, J., Guo, Y., Singh, S., and Lee, H. (2018). Self-imitation learning. In *Proceedings of the International Conference on Machine Learning*.

Oh, J., Singh, S., and Lee, H. (2017a). Value prediction network. In *Advances in the Neural Information Processing System*.

Oh, J., Singh, S., Lee, H., and Kohli, P. (2017b). Zero-shot task generalization with multi-task deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

Olton, D. S. (1979). Mazes, maps, and memory. *American psychologist*, 34(7):583.

Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. (2016). Deep exploration via boot-strapped dqn. In *Advances in the Neural Information Processing System*.

Ostrovski, G., Bellemare, M. G., Oord, A. v. d., and Munos, R. (2017). Count-based exploration with neural density models. In *Proceedings of the International Conference on Machine Learning*.

Parisotto, E., Ba, J. L., and Salakhutdinov, R. (2016). Actor-mimic: Deep multitask and transfer reinforcement learning. In *Proceedings of the International Conference on Learning Representations*.

Parisotto, E. and Salakhutdinov, R. (2018). Neural map: Structured memory for deep reinforcement learning. In *Proceedings of the International Conference on Learning Representations*.

Parr, R. and Russell, S. J. (1997). Reinforcement learning with hierarchies of machines. In *Advances in the Neural Information Processing System*.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the International Conference on Machine Learning*.

Precup, D. (2000). *Temporal abstraction in reinforcement learning*. PhD thesis, University of Massachusetts, Amherst.

Precup, D., Sutton, R. S., and Dasgupta, S. (2001). Off-policy temporal difference learning with function approximation. In *Proceedings of the International Conference on Machine Learning*.

Precup, D., Sutton, R. S., and Singh, S. (2000). Eligibility traces for off-policy policy evaluation. In *Proceedings of the International Conference on Machine Learning*.

Pritzel, A., Uria, B., Srinivasan, S., Puigdomènech, A., Vinyals, O., Hassabis, D., Wierstra, D., and Blundell, C. (2017). Neural episodic control. In *Proceedings of the International Conference on Machine Learning*.

Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Raiko, T. and Tornio, M. (2009). Variational bayesian learning of nonlinear hidden state-space models for model predictive control. *Neurocomputing*, 72(16):3704–3712.

Reed, S., Sohn, K., Zhang, Y., and Lee, H. (2014). Learning to disentangle factors of variation with manifold interaction. In *Proceedings of the International Conference on Machine Learning*.

Reed, S. E., Zhang, Y., Zhang, Y., and Lee, H. (2015). Deep visual analogy-making. In *Advances in the Neural Information Processing System*.

Rifai, S., Bengio, Y., Courville, A., Vincent, P., and Mirza, M. (2012). Disentangling factors of variation for facial expression recognition. In *Proceedings of the European Conference on Computer Vision*.

Ring, M. B. (1994). *Continual learning in reinforcement environments*. PhD thesis, University of Texas at Austin Austin.

Rusu, A. A., Colmenarejo, S. G., Gulcehre, C., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. (2016). Policy distillation. In *Proceedings of the International Conference on Learning Representations*.

Schaul, T., Horgan, D., Gregor, K., and Silver, D. (2015). Universal value function approximators. In *Proceedings of the International Conference on Machine Learning*.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. In *Proceedings of the International Conference on Learning Representations*.

Schmidhuber, J. (1991). Adaptive confidence and adaptive curiosity. In *Institut fur Informatik, Technische Universitat Munchen, Arcisstr. 21, 800 Munchen 2*.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Schmidhuber, J. and Huber, R. (1991). Learning to generate artificial fovea trajectories for target detection. *International Journal of Neural Systems*, 2:125–134.

Schulman, J., Chen, X., and Abbeel, P. (2017a). Equivalence between policy gradients and soft q-learning. *arXiv preprint arXiv:1704.06440*.

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

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017b). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

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.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the International Conference on Machine Learning*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017a). Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359.

Silver, D., Sutton, R. S., and Müller, M. (2012). Temporal-difference search in computer go. *Machine Learning*, 87:183–219.

Silver, D., van Hasselt, H., Hessel, M., Schaul, T., Guez, A., Harley, T., Dulac-Arnold, G., Reichert, D., Rabinowitz, N., Barreto, A., and Degris, T. (2017b). The predictron: End-to-end learning and planning. In *Proceedings of the International Conference on Machine Learning*.

Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations*.

Singh, S. (1991). The efficient learning of multiple task sequences. In *Advances in the Neural Information Processing System*.

Singh, S. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3-4):323–339.

Singh, S., Barto, A. G., and Chentanez, N. (2004). Intrinsically motivated reinforcement learning. In *Advances in the Neural Information Processing System*.

Singh, S., Litman, D., Kearns, M., and Walker, M. (2002). Optimizing dialogue management with reinforcement learning: Experiments with the njfun system. *Journal of Artificial Intelligence Research*, 16:105–133.

Srivastava, N., Mansimov, E., and Salakhutdinov, R. (2015). Unsupervised learning of video representations using LSTMs. In *Proceedings of the International Conference on Machine Learning*.

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

Stolle, M. and Precup, D. (2002). Learning options in reinforcement learning. In *Proceedings of International Symposium on Abstraction, Reformulation, and Approximation*.

Strehl, A. L. and Littman, M. L. (2008). An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, 74(8):1309–1331.

Sukhbaatar, S., Lin, Z., Kostrikov, I., Synnaeve, G., Szlam, A., and Fergus, R. (2018). Intrinsic motivation and automatic curricula via asymmetric self-play. In *Proceedings of the International Conference on Learning Representations*.

Sukhbaatar, S., Szlam, A., Synnaeve, G., Chintala, S., and Fergus, R. (2015a). Mazebase: A sandbox for learning from games. *arXiv preprint arXiv:1511.07401*.

Sukhbaatar, S., Weston, J., and Fergus, R. (2015b). End-to-end memory networks. In *Advances in the Neural Information Processing System*.

Sutskever, I., Hinton, G. E., and Taylor, G. W. (2009). The recurrent temporal restricted Boltzmann machine. In *Advances in the Neural Information Processing System*.

Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in the Neural Information Processing System*.

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

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in the Neural Information Processing System*.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Sutton, R. S., McAllester, D. A., Singh, S., and Mansour, Y. (1999a). Policy gradient methods for reinforcement learning with function approximation. In *Advances in the Neural Information Processing System*, volume 99.

Sutton, R. S., Precup, D., and Singh, S. (1999b). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211.

Sutton, R. S., Szepesvári, C., Geramifard, A., and Bowling, M. H. (2008). Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

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

Tamar, A., Levine, S., Abbeel, P., Wu, Y., and Thomas, G. (2016). Value iteration networks. In *Advances in the Neural Information Processing System*.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, O. X., Duan, Y., Schulman, J., DeTurck, F., and Abbeel, P. (2017). # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in the Neural Information Processing System*.

Taylor, G. W. and Hinton, G. E. (2009). Factored conditional restricted Boltzmann machines for modeling motion style. In *Proceedings of the International Conference on Machine Learning*.

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

Tellex, S., Knepper, R. A., Li, A., Rus, D., and Roy, N. (2014). Asking for help using inverse semantics. In *Robotics: Science and Systems*.

Tellex, S., Kollar, T., Dickerson, S., Walter, M. R., Banerjee, A. G., Teller, S. J., and Roy, N. (2011). Understanding natural language commands for robotic navigation and mobile manipulation. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

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

Tessler, C., Givony, S., Zahavy, T., Mankowitz, D. J., and Mannor, S. (2017). A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Thrun, S. B. (1992). The role of exploration in learning control. *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5 - RMSProp: Divde the gradient by a running average of its recent magnitude. *Coursera*.

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Tran, D., Bourdev, L., Fergus, R., Torresani, L., and Paluri, M. (2015). Learning spatiotemporal features with 3D convolutional networks. In *Proceedings of the International Conference on Computer Vision*.

van Hasselt, H. (2010). Double q-learning. In *Advances in the Neural Information Processing System*.

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

Vezhnevets, A., Mnih, V., Osindero, S., Graves, A., Vinyals, O., Agapiou, J., and Kavukcuoglu, K. (2016). Strategic attentive writer for learning macro-actions. In *Advances in the Neural Information Processing System*.

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 the International Conference on Machine Learning*.

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., et al. (2017). Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2017). Sample efficient actor-critic with experience replay. In *Proceedings of the International Conference on Machine Learning*.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning*.

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

Weber, T., Racanière, S., Reichert, D. P., Buesing, L., Guez, A., Rezende, D. J., Badia, A. P., Vinyals, O., Heess, N., Li, Y., Pascanu, R., Battaglia, P. W., Silver, D., and Wierstra, D. (2017). Imagination-augmented agents for deep reinforcement learning. In *Advances in the Neural Information Processing System*.

Weston, J., Chopra, S., and Bordes, A. (2015). Memory networks. In *Proceedings of the International Conference on Learning Representations*.

Wierstra, D., Förster, A., Peters, J., and Schmidhuber, J. (2010). Recurrent policy gradients. *Logic Journal of IGPL*, 18(5):620–634.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Xu, H., Gao, Y., Lin, J., Yu, F., Levine, S., and Darrell, T. (2018). Reinforcement learning from imperfect demonstrations. In *Proceedings of the International Conference on Machine Learning*.

Yang, J., Reed, S., Yang, M.-H., and Lee, H. (2015). Weakly-supervised disentangling with recurrent transformations for 3D view synthesis. In *Advances in the Neural Information Processing System*.

Yao, H., Bhatnagar, S., Diao, D., Sutton, R. S., and Szepesvári, C. (2009). Multi-step Dyna planning for policy evaluation and control. In *Advances in the Neural Information Processing System*.

Zaremba, W., Mikolov, T., Joulin, A., and Fergus, R. (2016). Learning simple algorithms from examples. In *Proceedings of the International Conference on Machine Learning*.

Zaremba, W. and Sutskever, I. (2016). Reinforcement learning neural turing machines. In *Proceedings of the International Conference on Learning Representations*.

Zhang, M., Levine, S., McCarthy, Z., Finn, C., and Abbeel, P. (2016). Policy learning with continuous memory states for partially observed robotic control. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Ziebart, B. D. (2010). *Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy*. PhD thesis, University of Washington.

Ziebart, B. D., Maas, A. L., Bagnell, J. A., and Dey, A. K. (2008). Maximum entropy inverse reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.