Toward Efficient Reinforcement Learning Under Non-Stationarity

by

Qingfeng Lan

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Non-stationarity in reinforcement learning (RL) arises naturally from value bootstrapping, policy iterations, and the resulting shifts in data distributions. These shifts lead to catastrophic forgetting, unstable training, and inefficient learning, which are further exacerbated by limited resources. This thesis addresses some of these challenges by developing novel algorithmic and architectural methods that reduce catastrophic forgetting, enhance learning efficiency, and improve training robustness under non-stationarity.

First, we propose memory-efficient RL algorithms that reduce reliance on large replay buffers by consolidating knowledge to mitigate forgetting, achieving strong performance within constrained memory budgets. Second, we investigate the role of neural network architecture in reducing forgetting and introduce elephant activation functions, which induce both sparse activations and gradients. This architectural modification significantly enhances resistance to forgetting and improves learning performance. Third, we explore the potential of learning optimizers tailored for RL, addressing challenges such as non-independent and identically distributed update gradients with high bias and variance. We present an effective meta-training framework that enables learned optimizers to generalize across diverse tasks. Finally, we present a novel method to exploit reward functions more effectively by incorporating reward gradients into policy gradient methods, without explicit dynamics modeling. This approach improves sample efficiency and performance, complementing the broader goal of efficient RL. Collectively, these contributions deepen the understanding of RL under non-stationary and resource-constrained conditions, paving the way toward more adaptable, efficient, and practical RL algorithms.

Preface

The chapters of this thesis are based on four papers. Specifically, Chapter 3 is based on Lan et al. (2023) which is accepted by Transactions on Machine Learning Research in 2023. Chapter 4 is based on Lan and Mahmood (2023), accepted by Workshop on High-dimensional Learning Dynamics of International Conference on Machine Learning in 2023. Moreover, Chapter 5 is based on Lan et al. (2024), accepted by Reinforcement Learning Journal in 2024. Finally, Chapter 6 and part of Chapter 2 are based on Lan et al. (2022) which is published on International Conference on Artificial Intelligence and Statistics in 2022. The aforementioned papers are listed below.

- **Qingfeng Lan**, A. Rupam Mahmood, Shuicheng Yan, Zhongwen Xu. "Learning to Optimize for Reinforcement Learning". *Reinforcement Learning Journal*, 2024.
- **Qingfeng Lan**, A. Rupam Mahmood. "Elephant Neural Networks: Born to Be a Continual Learner". *ICML Workshop on High-dimensional Learning Dynamics*, 2023.
- **Qingfeng Lan**, Yangchen Pan, Jun Luo, A. Rupam Mahmood. "Memory-efficient Reinforcement Learning with Value-based Knowledge Consolidation". *Transactions on Machine Learning Research*, 2023.
- **Qingfeng Lan**, Samuele Tosatto, Homayoon Farrahi, A. Rupam Mahmood. "Model-free Policy Learning with Reward Gradients". *International Conference on Artificial Intelligence and Statistics*, 2022.

Additionally, I have other publications not included in this thesis, which are listed below.

- Shibhansh Dohare, J. Fernando Hernandez-Garcia, **Qingfeng Lan**, Parash Rahman, A. Rupam Mahmood, Richard S. Sutton. "Loss of Plasticity in Deep Continual Learning". *Nature*,

2024.

- Mohamed Elsayed, **Qingfeng Lan**, Clare Lyle, A. Rupam Mahmood. "Weight Clipping for Deep Continual and Reinforcement Learning". *Reinforcement Learning Journal*, 2024.
- Haque Ishfaq, Yixin Tan, Yu Yang, **Qingfeng Lan**, Jianfeng Lu, A. Rupam Mahmood, Doina Precup, Pan Xu. "More Efficient Randomized Exploration for Reinforcement Learning via Approximate Sampling". *Reinforcement Learning Journal*, 2024.
- Haque Ishfaq*, **Qingfeng Lan***, Pan Xu, A. Rupam Mahmood, Doina Precup, Anima Anandkumar, Kamyar Azizzadenesheli. "Provable and Practical: Efficient Exploration in Reinforcement Learning via Langevin Monte Carlo". *International Conference on Learning Representations*, 2024.
- Shibhansh Dohare, **Qingfeng Lan**, A. Rupam Mahmood. "Overcoming Policy Collapse in Deep Reinforcement Learning". *European Workshop on Reinforcement Learning*, 2023.

To my families

All knowledge is, in final analysis, history.

All sciences are, in the abstract, mathematics.

All judgements are, in their rationale, statistics.

– Calyampudi Radhakrishna Rao

Acknowledgment

First and foremost, I am deeply grateful to my supervisor, Rupam Mahmood, for his invaluable guidance, critical insights, and continuous support throughout the course of my PhD. His intellectual rigor and professional standards have significantly shaped the direction and quality of my work.

I am also grateful to the members of my candidacy and thesis examining committees—Michael Bowling, Martha White, Marc'Aurelio Ranzato, Dale Schuurmans, and Pierre-Luc Bacon—for their valuable time, thoughtful engagement, and constructive feedback, all of which have significantly contributed to the development of this thesis. Next, I would like to thank my collaborators, co-authors, and internship mentors, whose expertise, support, and partnership have been integral to my PhD journey: Shibhansh Dohare, J. Fernando Hernandez-Garcia, Parash Rahman, Richard Sutton, Shuicheng Yan, Zhongwen Xu, Mohamed Elsayed, Clare Lyle, Haque Ishfaq, Yixin Tan, Yu Yang, Jianfeng Lu, Doina Precup, Pan Xu, Anima Anandkumar, Kamyar Azizzadenesheli, Yangchen Pan, Jun Luo, Samuele Tosatto, Homayoon Farrahi, Chao Gao, Rohan Chitnis, Alborz Geramifard, Ta-Chu Kao, and Jorge Menendez.

I acknowledge the financial support provided by the University of Alberta and Alberta Innovates, without which this research would not have been possible. I also acknowledge the broader contributions of the digital age—the open accessibility of information and resources on the Internet has been essential to both my research and daily life. On a personal note, I have found both relaxation and inspiration through videos, films, dramas, and anime on Bilibili.

Finally, I wish to thank my families and friends for their valuable support and encouragement throughout the ups and downs of this long journey.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and Objective

Deep reinforcement learning (RL) algorithms have shown great success in many applications, such as computer games (Vinyals et al. 2019, Badia et al. 2020, Schrittwieser et al. 2020, Zha et al. 2021, Wurman et al. 2022), simulated robotic tasks (Lillicrap et al. 2016a, Haarnoja et al. 2018, Hwangbo et al. 2019), and recommendation systems (Zheng et al. 2018, Zhao et al. 2018, Zhang et al. 2019b). However, the learning efficiency of these deep RL algorithms still remains limited. To achieve such success, these algorithms typically require substantial resources, such as millions of samples (Mnih et al. 2015), weeks of training (Berner et al. 2019), and hundreds of advanced GPUs (Vinyals et al. 2019). The requirement for large amounts of training resources makes RL less accessible to the general public, limiting its applicability in real-world settings—particularly for onboard and edge devices (Hayes et al. 2019, Hayes and Kanan 2022).

A key challenge that hinders the learning efficiency of RL is non-stationarity, which arises from value bootstrapping, policy iterations, and the resulting shifts in data distributions. The inherent non-stationarity in RL may lead to catastrophic forgetting (Ring 1994), poor generalization (Igl et al. 2021), low sample efficiency (Lyle et al. 2021), and prolonged training time to reach satisfactory performance (Dohare et al. 2024), especially when the resources are limited (Lan et al. 2023).

Among these issues, catastrophic forgetting remains to be one of the most critical challenges to achieving efficient learning for decades (McCloskey and Cohen 1989, French 1999, Wang et al. 2024). Specifically, it stands for the phenomenon that, when used with backpropagation, artificial neural networks tend to forget prior knowledge drastically, which is commonly encountered in both continual supervised learning (Hsu et al. 2018, Farquhar and Gal 2018, Van de Ven et al. 2022, Delange et al. 2021) and RL (Schwarz et al. 2018, Atkinson et al. 2021, Khetarpal et al. 2022). In RL, a learning agent exhibits catastrophic forgetting when it fails to retain past knowledge and overwrites previously learned skills during later training, resulting in deteriorating performance and inefficient learning (Ghiassian et al. 2020, Pan et al. 2022a).

Besides forgetting, learning under non-stationarity also makes RL a difficult optimization problem with low learning efficiency. For example, in value-based methods, temporal-difference (TD) is widely applied in value iterations:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)),$$

where α is the learning rate, s_t and s_{t+1} are two successive states, and $r_{t+1} + \gamma V(s_{t+1})$ is named the TD target. TD targets are usually biased, non-independent and identically distributed (non-IID), and noisy, due to changing state-values, complex state transitions, and noisy reward signals (Schulman et al. 2016), inducing a changing loss landscape that evolves during training. As a result, in value-based methods, the gradients usually have high bias and variance which lead to sub-optimal performance or even a failure of convergence. Similarly, in policy gradient methods, the gradients of the objective function (see Equation (2.8) in Section 2.2.2 for more details) are also notoriously high in variance. In summary, unstable gradients in RL training result in difficult optimization and inefficient learning (Zhao et al. 2019).

In this thesis, we aim to improve the learning efficiency of RL under non-stationarity, thereby reducing the need for extensive training resources. Specifically, we define learning efficiency in terms of sample usage, memory consumption, and computational cost.

1.2 Approaches and Contributions

To improve the learning efficiency of RL under non-stationarity, this thesis tackles some of the challenges through a combination of algorithmic innovation, architectural design, and meta-learning techniques. In the following, we briefly summarize our contributions in this thesis.

Memory-Efficient Reinforcement Learning Algorithms To reduce forgetting and improve sample efficiency, the experience replay buffer is widely applied and it has become a standard component in deep RL. Typically, experiences are stored in a large buffer and used for training at a later stage. However, a large replay buffer results in a heavy memory burden, especially for onboard and edge devices with limited memory capacities. We propose memory-efficient RL algorithms based on the deep Q-network algorithm to alleviate this issue. Our algorithms reduce forgetting and maintain high sample efficiency by consolidating knowledge from the target Q-network to the current Q-network. Compared to baselines, our algorithms achieve comparable or better performance in both feature-based and image-based tasks while easing the burden of large experience replay buffers.

A Novel Activation Function that Helps Reduce Forgetting Recent works have proposed effective methods to mitigate forgetting; however, these efforts primarily focus on algorithmic solutions. Meanwhile, we do not have a well-understood account of what architectural properties of neural networks lead to catastrophic forgetting. We aim to fill this gap by studying the role of activation functions in the training dynamics of neural networks and their impact on catastrophic forgetting in RL setup. Our study reveals that, besides sparse representations, the gradient sparsity of activation functions also plays an important role in reducing forgetting. Based on this insight, we propose a new class of activation functions, elephant activation functions, that can generate both sparse outputs and sparse gradients. We show that by simply replacing classical activation functions with elephant activation functions in the neural networks of value-based algorithms, we can significantly improve the resilience of neural networks to catastrophic forgetting, thus making RL algorithms more sample-efficient and memory-efficient.

A Learned Optimizer Designed for Reinforcement Learning To alleviate the need for manually designing optimizers, researchers have proposed using meta-learning to automatically learn them. In recent years, by leveraging more data, computation, and diverse tasks, learned optimizers have achieved remarkable success in supervised learning, outperforming classical hand-designed optimizers. RL is essentially different from supervised learning, and in practice, these learned optimizers do not work well even in simple RL tasks. We investigate this phenomenon and identify two issues that are related to the non-stationarity of RL. First, the agent-gradient distribution is non-independent and identically distributed, leading to inefficient meta-training. Moreover, due to highly stochastic agent-environment interactions, the agent-gradients have high bias and variance, which increases the difficulty of learning an optimizer for RL. We propose pipeline training and a novel optimizer structure with a good inductive bias to address these issues, making it possible to learn an optimizer for RL from scratch. We show that, although only trained in toy tasks, our learned optimizer can generalize to unseen complex tasks.

An Innovative Policy Gradient Method that Integrates Reward Gradients Despite the increasing popularity of policy gradient methods, they are yet to be widely utilized in sample-scarce applications, such as robotics. The sample efficiency could be improved by making best usage of available information. As a key component in RL, the reward function is usually devised carefully to guide the agent. Hence, the reward function is usually known, allowing access to not only scalar reward signals but also reward gradients. To benefit from reward gradients, previous works require the knowledge of environment dynamics, which is hard to obtain. In this work, we develop the reward policy gradient estimator, a novel approach that integrates reward gradients without learning a model. Bypassing the model dynamics allows our estimator to achieve a better bias-variance trade-off, which results in a higher sample efficiency, as shown in the empirical analysis. Our method also boosts the performance of PPO on different control tasks.

In summary, this thesis systematically investigates the challenges posed by non-stationarity in RL and proposes practical, resource-efficient solutions across algorithmic, architectural, and optimization dimensions. The presented approaches aim to advance the development of more accessible,

scalable, and efficient RL systems, particularly for real-world deployment under constrained computational resources.

Chapter 2

Background

In this chapter, we introduce the basic concepts of reinforcement learning, a few classical reinforcement learning algorithms with related theories, as well as the background of catastrophic forgetting, knowledge distillation, and meta-gradient methods.

2.1 Markov Decision Process

Reinforcement learning (RL) is a computational approach to goal-directed learning from interactions with an environment (Sutton and Barto 2018). To achieve a goal, a learning agent must be capable of perceiving the state of the environment and executing actions that affect the state.

Specifically, the problems of RL can be formalized as Markov decision processes (MDPs). Let $\Delta(X)$ be the space of all probability distributions supported over the set X. Consider a MDP, $M = (\mathcal{S}, \mathcal{A}, \mathrm{P}, p_0, R, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathrm{P} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the transition probability, $p_0 \in \Delta(\mathcal{S})$ is the initial state distribution, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1]$ is the discount factor.

An agent interacts with the MDP environment based on a policy $\pi \in \Pi : \mathcal{S} \to \Delta(\mathcal{A})$. Specifically, the agent starts from state $s_0 \sim p_0(\cdot)$. At each time-step t, it observes the state $s_t \in \mathcal{S}$, takes an action $a_t \sim \pi(\cdot|s_t)$, transits to the next state $s_{t+1} \sim \mathrm{P}(\cdot|s_t, a_t)$, and receives a scalar reward

$r_{t+1} = R(s_t, a_t)$. A trajectory (up to time-step T) is defined as $\tau = (s_0, a_0, r_1, s_1, \cdots, s_T)$. Define return G_t over τ as the total (discounted) reward from time-step t:

$$G_t \doteq \sum_{i=t}^{T-1} \gamma^{i-t} R(s_i, a_i). \tag{2.1}$$

State-value functions are defined as the expected return under policy π,

$$V_\pi(s) \doteq \mathbb{E}_\pi[G_t | s_t = s]. \tag{2.2}$$

Similarly, action-value functions are defined as

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | s_t = s, a_t = a]. \tag{2.3}$$

Furthermore, by definitions V_π and Q_π are connected with the following equations:

$$\begin{aligned} V_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a), \\ Q_\pi(s, a) &= R(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathrm{P}(s'|s, a) V_\pi(s'). \end{aligned}$$

With the above equations, we can compute V_π and Q_π recursively:

$$\begin{aligned} V_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a) \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) [R(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathrm{P}(s'|s, a) V_\pi(s')], \qquad (2.4) \\ Q_\pi(s, a) &= R(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathrm{P}(s'|s, a) V_\pi(s') \\ &= R(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathrm{P}(s'|s, a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s', a'). \qquad (2.5) \end{aligned}$$

Specifically, Equation (2.4) and Equation (2.5) are known as the Bellman equations.

The goal of the agent is to find an optimal policy π_* such that $V_{\pi_*}(s) \geq V_\pi(s)$ for all $s \in \mathcal{S}$ and

$\pi \in \Pi$. By definition, all optimal policies should have the same value functions, called the optimal value functions, which are formally defined as

$$V_*(s) = \max_\pi V_\pi(s),$$
$$Q_*(s,a) = \max_\pi Q_\pi(s,a).$$

Similar to value functions, V_* and Q_* are also connected:

$$\begin{aligned} V_*(s) =& \max_{a\in\mathcal{A}} Q_*(s,a), \\ Q_*(s,a) =& R(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathrm{P}(s'|s,a) V_*(s'). \end{aligned}$$

Furthermore, for optimal value functions we have the Bellman optimality equations:

$$\begin{aligned} V_*(s) =& \max_{a\in\mathcal{A}} Q_*(s,a), \\ =& \max_{a\in\mathcal{A}} \left[R(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathrm{P}(s'|s,a) V_*(s') \right], \qquad (2.6) \\ Q_*(s,a) =& R(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathrm{P}(s'|s,a) V_*(s') \\ =& R(s,a) + \gamma \sum_{s'\in\mathcal{S}} \mathrm{P}(s'|s,a) \max_{a'\in\mathcal{A}} Q_*(s',a'). \qquad (2.7) \end{aligned}$$

2.2 Reinforcement Learning

Based on a policy function is learned, RL algorithms can mainly be divided into two categories: value-based methods and policy gradient methods. In the following sections, we explain them one by one.

2.2.1 Value-Based Methods

In value-based methods, we do not learn a policy function directly; instead, we train an action-value function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ first and then extract a policy given this action-value function.

Among them, Q-learning (Watkins 1989) is one of the most popular value-based algorithms. Concretely, it applies temporal-difference (TD) learning to learn the optimal action-value function. For a sampled transition $\{s_t, a_t, r_{t+1}, s_{t+1}\}$, the updating rule is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(Y - Q(s_t, a_t)),$$

where α is learning rate and $Y \doteq r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')$ is the target action-value adapted from the Bellman optimality equation (Equation (2.7)). The corresponding greedy policy is extracted from Q and formally defined as

$$\pi_{\text{greedy}}(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q(s, a') \\ 0, & \text{otherwise} \end{cases}.$$

With a slight abuse of notation, we can also write $\pi_{\text{greedy}}(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$. To strike a better exploration-exploitation balance, we usually employ the ϵ-greedy policy:

$$\pi_{\epsilon\text{-greedy}}(a|s) = \frac{\epsilon}{|\mathcal{A}|} + \begin{cases} 1 - \epsilon, & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q(s, a') \\ 0, & \text{otherwise} \end{cases},$$

where $0 \leq \epsilon \leq 1$. Essentially, ϵ-greedy policy adds a certain exploration noise controlled by ϵ to the greedy policy. At each time step, the ϵ-greedy policy selects the greedy action (i.e., the action with the highest action-value) with probability $1 - \epsilon$, or a random action (uniformly at random over all possible actions) with probability ϵ.

2.2.2 Policy Gradient Methods

Unlike value-based methods, policy gradient methods aim to learn and approximate an optimal policy function directly. Among them, besides learning a policy function (i.e., the actor), actor-critic algorithms also approximate a value function (i.e., the critic) to help learning.

For convenience, let's consider continuous state and action spaces.[1] The goal of the agent is to obtain a policy π that maximizes the expected return starting from the initial states. Let a policy π_θ be a differentiable function of a weight vector θ. Our goal is to find θ that maximizes

$$J(\theta) = \int p_0(s) V_{\pi_\theta}(s) \mathrm{d}s. \tag{2.8}$$

To this end, we can apply gradient ascent techniques. Since the true gradient $\nabla_\theta J(\theta)$ is not typically available, we resort to Monte Carlo methods (Mohamed et al. 2020). This gradient estimation problem can be formalized as computing the unbiased gradient of the expectation of a function with respect to some parameters of a distribution. Specifically, let $p_\theta(x)$ be the probability distribution of x with parameters θ. Define $F(\theta) = \int p_\theta(x)\phi(x)\mathrm{d}x$. Then the key step of the problem can be formally defined as estimating $\nabla_\theta F(\theta)$:

$$\nabla_\theta F(\theta) = \nabla_\theta \mathbb{E}_{X \sim p_\theta}[\phi(X)].$$

The gradient estimation problem is fundamental in many machine learning areas, such as reinforcement learning (Williams 1992), variational inference (Hoffman et al. 2013), evolutionary algorithms (Conti et al. 2018), and variational auto-encoders (Kingma and Welling 2013). In general, there are many approaches for gradient estimation, such as likelihood-ratio gradient estimators, reparameterization gradient estimators, finite difference methods, infinitesimal perturbation analysis (L'ecuyer 1990), and mean-valued derivative methods (Pflug 1989, Carvalho et al. 2021). Next, we introduce two major approaches for solving this problem in RL, resulting in various policy gradient algorithms.

[1] For discrete state and action spaces, we replace summation with integration for suitable equations in the following.

Likelihood-Ratio Gradient Estimators

The likelihood-ratio (LR) gradient estimator (Glynn 1990) is one of the most popular gradient estimators. This estimator applies the log-derivative technique $\nabla_\theta \log p_\theta(x) = \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)}$ to obtain the unbiased gradient estimation:

$$\begin{aligned}
&\nabla_\theta F(\theta) \\
=&\nabla_\theta \mathbb{E}_{p_\theta(x)}[\phi(x)] = \nabla_\theta \int p_\theta(x)\phi(x)\mathrm{d}x \\
=&\int \phi(x)\nabla_\theta p_\theta(x)\mathrm{d}x = \int \phi(x) p_\theta(x)\nabla_\theta \log p_\theta(x)\mathrm{d}x \\
=&\mathbb{E}_{X\sim p_\theta}[\phi(X)\nabla_\theta \log p_\theta(X)].
\end{aligned} \tag{2.9}$$

The LR estimator is a fundamental component of the policy gradient theorem (Sutton and Barto 2018). Partly because of its generality of being applicable to both continuous and discrete action spaces, the LR estimator has been used in many policy gradient methods. In the following, we show a direct application of the LR estimator in RL and provide a proof of the policy gradient theorem for completeness.

First, we make two common assumptions on the MDP following Imani et al. (2018).

Assumption 2.1. *$\mathcal{S}$ and $\mathcal{A}$ are closed and bounded.*

Assumption 2.2. *$\mathrm{P}(s'|s,a)$, $\pi_\theta(a|s)$, $R(s,a)$, $p_0(s)$ and their derivatives are continuous in variables s, a, s', and θ.*

The two assumptions above allow us to exchange derivatives and integrals, and the order of multiple integrations, using Fubini's theorem and Leibniz integral rule. Now, we are ready to move on to the proof.

Theorem 2.1 (Policy Gradient Theorem)**.** *Suppose that the MDP satisfies Assumption 2.1 and Assumption 2.2, then*

$$\nabla_\theta J(\theta) = \int d^{\pi_\theta,\gamma}(s)\pi_\theta(a|s)Q_{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s.$$

where $d^{\pi_\theta,\gamma}(s') = \int \sum_{t=0}^{\infty} \gamma^t p_0(s) p(s \to s', t, \pi_\theta) \mathrm{d}s$ *is the (discounted) stationary state distribution under* π_θ *and* $p(s \to s', t, \pi_\theta)$ *is the transition probability from* s *to* s' *with* t *steps under* π_θ.

Proof. By definition, we have

$$\begin{aligned} V_{\pi_\theta}(s) =& \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t) \mid s_0 = s \right], \\ Q_{\pi_\theta}(s, a) =& \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s, a_0 = a \right]. \end{aligned}$$

Then V_{π_θ} and Q_{π_θ} are connected by

$$\begin{aligned} Q_{\pi_\theta}(s, a) =& R(s, a) + \gamma \int \mathrm{P}(s'|s, a) V_{\pi_\theta}(s') \mathrm{d}s', \\ V_{\pi_\theta}(s) =& \int \pi_\theta(a|s) Q_{\pi_\theta}(s, a) \mathrm{d}a. \end{aligned}$$

Thus

$$\begin{aligned} & \nabla_\theta V_{\pi_\theta}(s) \\ =& \nabla_\theta \left(\int \pi_\theta(a|s) Q_{\pi_\theta}(s, a) \mathrm{d}a \right) \\ =& \int Q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s) \mathrm{d}a + \int \pi_\theta(a|s) \nabla_\theta Q_{\pi_\theta}(s, a) \mathrm{d}a \\ =& \int Q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s) \mathrm{d}a + \int \pi_\theta(a|s) \nabla_\theta \left(R(s, a) + \gamma \int \mathrm{P}(s'|s, a) V_{\pi_\theta}(s') \mathrm{d}s' \right) \mathrm{d}a \\ =& \int Q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s) \mathrm{d}a + \gamma \int \mathrm{P}(s'|s, a) \pi_\theta(a|s) \nabla_\theta V_{\pi_\theta}(s') \mathrm{d}s' \mathrm{d}a \\ =& \int Q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s) \mathrm{d}a + \gamma \int p(s \to s', 1, \pi_\theta) \nabla_\theta V_{\pi_\theta}(s') \mathrm{d}s', \end{aligned}$$

where $p(s \to s', 1, \pi_\theta) = \int \pi_\theta(a|s) \mathrm{P}(s'|s, a) \mathrm{d}a$.

Now, iterating this formula, we have

$$\nabla_\theta V_{\pi_\theta}(s)$$

$$
\begin{aligned}
&= \int Q_{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s)\mathrm{d}a + \gamma \int p(s \to s', 1, \pi_\theta)\nabla_\theta V_{\pi_\theta}(s')\mathrm{d}s' \\
&= \int Q_{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s)\mathrm{d}a + \gamma \int p(s \to s', 1, \pi_\theta) \\
&\quad \left(\int Q_{\pi_\theta}(s',a')\nabla_\theta \pi_\theta(a'|s')\mathrm{d}a' + \gamma \int p(s' \to s'', 1, \pi_\theta)\nabla_\theta V_{\pi_\theta}(s'')\mathrm{d}s'' \right) \mathrm{d}s' \\
&= \int Q_{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s)\mathrm{d}a + \gamma \int p(s \to s', 1, \pi_\theta) Q_{\pi_\theta}(s',a')\nabla_\theta \pi_\theta(a'|s')\mathrm{d}a'\mathrm{d}s' \\
&\quad + \gamma^2 \int p(s \to s', 1, \pi_\theta) p(s' \to s'', 1, \pi_\theta)\nabla_\theta V_{\pi_\theta}(s'')\mathrm{d}s''\mathrm{d}s' \\
&= \int Q_{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s)\mathrm{d}a + \gamma \int p(s \to s', 1, \pi_\theta) Q_{\pi_\theta}(s',a')\nabla_\theta \pi_\theta(a'|s')\mathrm{d}a'\mathrm{d}s' \\
&\quad + \gamma^2 \int p(s \to s', 2, \pi_\theta)\nabla_\theta V_{\pi_\theta}(s')\mathrm{d}s' \\
&= \dots \\
&= \int \sum_{t=0}^{\infty} \gamma^t p(s \to s', t, \pi_\theta) \left(\int Q_{\pi_\theta}(s',a')\nabla_\theta \pi_\theta(a'|s')\mathrm{d}a' \right) \mathrm{d}s',
\end{aligned}
$$

where $p(s \to s'', t+1, \pi_\theta) = \int p(s \to s', t, \pi_\theta) p(s' \to s'', 1, \pi_\theta)\mathrm{d}s'$.

Finally,

$$
\begin{aligned}
&\nabla_\theta J(\theta) \\
=&\nabla_\theta \int p_0(s) V_{\pi_\theta}(s)\mathrm{d}s = \int p_0(s)\nabla_\theta V_{\pi_\theta}(s)\mathrm{d}s \\
=& \int \sum_{t=0}^{\infty} \gamma^t p_0(s) p(s \to s', t, \pi_\theta) Q_{\pi_\theta}(s',a')\nabla_\theta \pi_\theta(a'|s')\mathrm{d}a'\mathrm{d}s'\mathrm{d}s \\
=& \int d^{\pi_\theta,\gamma}(s) Q_{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s \\
=& \int d^{\pi_\theta,\gamma}(s)\pi_\theta(a|s) Q_{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s, \quad \text{(Equation (2.9))}
\end{aligned}
$$

where $d^{\pi_\theta,\gamma}(s') = \int \sum_{t=0}^{\infty} \gamma^t p_0(s) p(s \to s', t, \pi_\theta)\mathrm{d}s$. □

Although the above theorem suggests sampling from the discounted stationary state distribution $d^{\pi_\theta,\gamma}$, in practice sampling is usually done from the related undiscounted stationary state distribution or a replay buffer. We denote this undiscounted stationary state distribution as $d^{\pi_\theta} = d^{\pi_\theta,\gamma=1}$. For a more detailed discussion, please check Zhang et al. (2022).

Moreover, variance reduction (Greensmith et al. 2004) is usually necessary to fully exert the power of the LR estimator. Subtracting a baseline (Williams 1992), applying eligibility traces (Singh and Sutton 1996), and utilizing the generalized advantage estimator (GAE) (Schulman et al. 2016) are three effective approaches to mitigate the variance issue of policy gradient based on the LR estimator. In the following, we show that the gradient $\nabla_\theta J(\theta)$ is still unbiased after subtracting the state-value baseline V_{π_θ}.

Corollary 2.1 (Policy Gradient Theorem with the State-Value Baseline)**.** *Suppose that the MDP satisfies Assumption 2.1 and Assumption 2.2, then*

$$\nabla_\theta J(\theta) = \int d^{\pi_\theta,\gamma}(s)\pi_\theta(a|s)(Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s))\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s.$$

where $d^{\pi_\theta,\gamma}(s') = \int \sum_{t=0}^{\infty} \gamma^t p_0(s) p(s \to s', t, \pi_\theta)\mathrm{d}s$ *is the (discounted) stationary state distribution under* π_θ *and* $p(s \to s', t, \pi_\theta)$ *is the transition probability from* s *to* s' *with* t *steps under* π_θ*.*

Proof. Considering Theorem 2.1, we only need to prove that

$$\int d^{\pi_\theta,\gamma}(s)\pi_\theta(a|s)V_{\pi_\theta}(s)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s = 0.$$

Specifically,

$$\begin{aligned}
&\int d^{\pi_\theta,\gamma}(s)\pi_\theta(a|s)V_{\pi_\theta}(s)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s \\
=&\int d^{\pi_\theta,\gamma}(s)V_{\pi_\theta}(s)\left(\int \pi_\theta(a|s)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\right)\mathrm{d}s \\
=&\int d^{\pi_\theta,\gamma}(s)V_{\pi_\theta}(s)\left(\int \nabla_\theta \pi_\theta(a|s)\mathrm{d}a\right)\mathrm{d}s \\
=&\int d^{\pi_\theta,\gamma}(s)V_{\pi_\theta}(s)\left(\nabla_\theta \int \pi_\theta(a|s)\mathrm{d}a\right)\mathrm{d}s \\
=&\int d^{\pi_\theta,\gamma}(s)V_{\pi_\theta}(s)\left(\nabla_\theta 1\right)\mathrm{d}s = 0.
\end{aligned}$$

This completes the proof. □

Many actor-critic algorithms use this LR estimator to estimate gradient, such as asynchronous advantage actor-critic (Mnih et al. 2016), trust region policy optimization (Schulman et al. 2015), proximal policy optimization (PPO) (Schulman et al. 2017), and actor-critic with experience replay (Wang et al. 2017).

Reparameterization Gradient Estimators

The reparameterization (RP) gradient estimator is also known as the pathwise gradient estimator or the reparameterization technique (Mohamed et al. 2020). Given the underlying probability distribution $p_\theta(x)$, this estimator takes the advantage of the knowledge of distribution $p_\theta(x)$ and reparameterize $p_\theta(x)$ with a simpler base distribution $p(\epsilon)$ that makes two equivalent sampling processes:

$$X \sim p_\theta(\cdot) \iff X = f_\theta(\epsilon),\ \epsilon \sim p(\cdot), \tag{2.10}$$

where f_θ is a function that maps ϵ to x. In other words, there are two equivalent ways to sample $X \sim p_\theta(x)$: one is to sample it directly; the other way is to first sample ϵ from a base distribution $p(\epsilon)$ and then apply a function f to transform ϵ to X. For example, assume X is a Gaussian variable where $X \sim \mathcal{N}(\mu, \sigma)$ and $\theta = [\mu, \sigma]$. Let the base distribution be $p(\epsilon) = \mathcal{N}(0, 1)$. Then X can be reparameterized as $X = f_\theta(\epsilon) = \mu + \sigma\epsilon$. For many common continuous distributions (e.g., the Gaussian, Log-Normal, Exponential, and Laplace), there exists such ways to reparameterize them with a simpler base distribution. Finally, we can write the gradient estimation as

$$\nabla_\theta F(\theta) = \nabla_\theta \int p_\theta(x)\phi(x)\mathrm{d}x = \nabla_\theta \int p(\epsilon)\phi(f_\theta(\epsilon))\mathrm{d}\epsilon = \int p(\epsilon)\nabla_\theta \phi(f_\theta(\epsilon))\mathrm{d}\epsilon. \tag{2.11}$$

Note that $p_\theta(x) = p(\epsilon)|\nabla_\epsilon f_\theta(\epsilon)|^{-1}$ due to integration by substitution. RP estimators are only applicable to known and continuous distributions.[2] In general, there is no guarantee that RP estimators outperform LR estimators (Gal 2016, Parmas et al. 2018). However, RP estimators

[2] An application to discrete random variables is possible by reparameterizing the Gumbel distribution—the continuous counterpart of the categorical distribution (Jang et al. 2017).

have a lower variance under certain assumptions (Xu et al. 2019), which usually lead to great benefits in many areas (Mohamed et al. 2020). Kingma and Welling (2013) applied the RP estimator to obtain a differentiable estimator of the variational lower bound that can be optimized directly using standard stochastic gradient methods. Rezende et al. (2014) used RP estimators for deep generative models and proposed deep latent Gaussian models that generate realistic images.

In many RL algorithms, RP estimators play a crucial role in reparameterizing actions and decoupling the randomness from a policy, such as soft actor-critic (SAC) (Haarnoja et al. 2018), SVG (Heess et al. 2015), and RELAX (Grathwohl et al. 2018). Deterministic policy gradient algorithm (DPG) (Silver et al. 2014), deep deterministic policy gradient algorithm (DDPG) (Lillicrap et al. 2016a), and deterministic value-policy gradient algorithm (Cai et al. 2020b) can also be viewed as special cases of these algorithms, where the probability density function of the base distribution is a Dirac delta function. Finally, Wang et al. (2019) proposed a class of RL algorithms called reparameterizable RL, where the randomness of the environment is decoupled from the trajectory distribution via the reparameterization technique.

In the following, we prove the reparameterization policy gradient theorem which directly applies the RP estimator to compute the gradient of the policy objective (Equation (2.8)).

We begin by assuming that the action a is sampled from the policy π parameterized with θ given the current state s: $a \sim \pi_\theta(\cdot|s)$. We then reparameterize the policy with a function f: $a = f_\theta(\epsilon; s)$, $\epsilon \sim p(\cdot)$. Let function g be the inverse function of f, that is, $\epsilon = g_\theta(a; s)$ and $a = f_\theta(g_\theta(a; s); s)$. Furthermore, similar to Assumption 2.2, we make one more assumption on the MDP in the following.

Assumption 2.3. *$f_\theta(\epsilon; s)$, $g_\theta(a|s)$, $p(\epsilon)$, and their derivatives are continuous in variables s, a, θ, and ϵ.*

Theorem 2.2 (Reparameterization Policy Gradient Theorem)**.** *Suppose that the MDP satisfies Assumption 2.1, Assumption 2.2, and Assumption 2.3, then*

$$\nabla_\theta J(\theta) = \int d^{\pi_\theta,\gamma}(s) p(\epsilon) \nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} \mathrm{d}\epsilon \mathrm{d}s.$$

Proof. By Theorem 2.1, we have

$$\nabla_\theta J(\theta) = \int d^{\pi_\theta,\gamma}(s)\pi_\theta(a|s)Q_{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s.$$

Thus,

$$\begin{aligned}
&\nabla_\theta J(\theta)\\
=&\int d^{\pi_\theta,\gamma}(s)\pi_\theta(a|s)Q_{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s\\
=&\int d^{\pi_\theta,\gamma}(s)\left(\int Q_{\pi_\theta}(s,a)\nabla_\theta\pi_\theta(a|s)\mathrm{d}a\right)\mathrm{d}s\\
=&\int d^{\pi_\theta,\gamma}(s)\left[\int \nabla_\theta\left(Q_{\pi_\theta}(s,a)\pi_\theta(a|s)\right)\mathrm{d}a - \int \pi_\theta(a|s)\nabla_\theta Q_{\pi_\theta}(s,a)\mathrm{d}a\right]\mathrm{d}s\\
=&\int d^{\pi_\theta,\gamma}(s)\left[\nabla_\theta\left(\int Q_{\pi_\theta}(s,a)\pi_\theta(a|s)\mathrm{d}a\right) - \int \pi_\theta(a|s)\nabla_\theta Q_{\pi_\theta}(s,a)\mathrm{d}a\right]\mathrm{d}s\\
=&\int d^{\pi_\theta,\gamma}(s)\left[\nabla_\theta\left(\int p(\epsilon)Q_{\pi_\theta}(s,f_\theta(\epsilon;s))\mathrm{d}\epsilon\right) - \int p(\epsilon)\nabla_\theta Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}\mathrm{d}\epsilon\right]\mathrm{d}s \quad \text{(by reparameterization)}\\
=&\int d^{\pi_\theta,\gamma}(s)\left[\int p(\epsilon)\left(\nabla_\theta f_\theta(\epsilon;s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)} + \nabla_\theta Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}\right)\mathrm{d}\epsilon\right.\\
&\left. - \int p(\epsilon)\nabla_\theta Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}\mathrm{d}\epsilon\right]\mathrm{d}s\\
=&\int d^{\pi_\theta,\gamma}(s)\ p(\epsilon)\nabla_\theta f_\theta(\epsilon;s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}\mathrm{d}\epsilon\mathrm{d}s.
\end{aligned}$$

□

Next, we introduce the deterministic policy gradient theorem, which is first proved by Silver et al. (2014). Moreover, both DPG and DDPG are proposed based on this theorem. It can also be proved as a corollary of Theorem 2.2 by considering the settings of deterministic policies.

Corollary 2.2 (Deterministic Policy Gradient Theorem)**.** *Suppose that the MDP satisfies Assumption 2.1, Assumption 2.2 and Assumption 2.3, for a deterministic policy $\mu_\theta(s)$, we have*

$$\nabla_\theta J(\theta) = \int d^{\mu_\theta,\gamma}(s)\nabla_\theta\mu_\theta(s)\nabla_a Q_{\mu_\theta}(s,a)|_{a=\mu_\theta(s)}\mathrm{d}s.$$

Proof. Let $p(\epsilon)$ be the delta function. Thus $\int_\epsilon p(\epsilon)f_\theta(\epsilon;s)\mathrm{d}\epsilon = f_\theta(0;s)$. Furthermore, let $f_\theta(0;s) =$

$\mu_\theta(s)$. By Theorem 2.2,

$$\begin{aligned}
&\nabla_\theta J(\theta) \\
=&\int d^{\pi_\theta,\gamma}(s)\left(\int p(\epsilon)\nabla_\theta f_\theta(\epsilon;s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}\mathrm{d}\epsilon\right)\mathrm{d}s \\
=&\int d^{\pi_\theta,\gamma}(s)\nabla_\theta f_\theta(0;s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=f_\theta(0;s)}\mathrm{d}s \\
=&\int d^{\pi_\theta,\gamma}(s)\nabla_\theta \mu_\theta(s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=\mu_\theta(s)}\mathrm{d}s.
\end{aligned}$$

□

Finally, we introduce the entropy-regularized reparameterization policy gradient theorem, which can also be derived as a corollary of Theorem 2.2.

Corollary 2.3 (Entropy-Regularized Reparameterization Policy Gradient Theorem)**.** *Consider entropy-regularized value functions*

$$\begin{aligned}
V_{\pi_\theta}(s) =&\mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t(R(s_t,a_t)+\alpha\mathcal{H}(\pi_\theta(\cdot|s_t)))\mid s_0=s\right], \\
Q_{\pi_\theta}(s,a) =&\mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t)+\alpha\sum_{t=1}^{\infty}\gamma^t\mathcal{H}(\pi_\theta(\cdot|s_t))\mid s_0=s,a_0=a\right],
\end{aligned}$$

where $\mathcal{H}(p)=-\int_x p(x)\log p(x)\mathrm{d}x$ is the differential entropy for probability density function $p(x)$, and α is a positive constant. Suppose that the MDP satisfies Assumption 2.1, Assumption 2.2 and Assumption 2.3, then

$$\nabla_\theta J(\theta)=\int d^{\pi_\theta,\gamma}(s)p(\epsilon)\left[\nabla_\theta f_\theta(\epsilon;s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}-\alpha\nabla_\theta\log\pi_\theta(f_\theta(\epsilon;s)|s)\right]\mathrm{d}\epsilon\mathrm{d}s.$$

Proof. Given the definitions of $V_{\pi_\theta}(s)$ and $Q_{\pi_\theta}(s,a)$, easy to verify that they are connected by

$$\begin{aligned}
&V_{\pi_\theta}(s) \\
=&\mathbb{E}_\pi[Q_{\pi_\theta}(s,a)]+\alpha\mathcal{H}(\pi_\theta(\cdot|s)) \\
=&\int\pi_\theta(a|s)(Q_{\pi_\theta}(s,a)-\alpha\log\pi_\theta(a|s))\mathrm{d}a
\end{aligned}$$

$$= \int p(\epsilon) \left(Q_{\pi_\theta}(s, f_\theta(\epsilon; s)) - \alpha \log \pi_\theta(f_\theta(\epsilon; s)|s) \right) \mathrm{d}\epsilon.$$

The Bellman equation for Q_{π_θ} is

$$\begin{aligned}
&Q_{\pi_\theta}(s, a) \\
=&\mathbb{E}_\pi \left[R(s, a) + \gamma (Q_{\pi_\theta}(s', a') + \alpha \mathcal{H}(\pi_\theta(\cdot|s'))) \right] \\
=&R(s, a) + \gamma \mathbb{E}[V_{\pi_\theta}(s')] = R(s, a) + \gamma \int \mathrm{P}(s'|s, a) V_{\pi_\theta}(s') \mathrm{d}s'.
\end{aligned}$$

Then

$$\begin{aligned}
&\nabla_\theta V_{\pi_\theta}(s) \\
=&\nabla_\theta \left(\int \pi_\theta(a|s) Q_{\pi_\theta}(s, a) \mathrm{d}a \right) + \alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) \\
=&\nabla_\theta \left(\int p(\epsilon) Q_{\pi_\theta}(s, f_\theta(\epsilon; s)) \mathrm{d}\epsilon \right) + \alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) \\
=&\int p(\epsilon) \nabla_\theta Q_{\pi_\theta}(s, f_\theta(\epsilon; s)) \mathrm{d}\epsilon + \alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) \\
=&\int p(\epsilon) \left(\nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} + \nabla_\theta Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} \right) \mathrm{d}\epsilon + \alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) \\
=&\int p(\epsilon) \left(\nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} + \gamma \int \mathrm{P}(s'|s, f_\theta(\epsilon; s)) \nabla_\theta V_{\pi_\theta}(s') \mathrm{d}s' \right) \mathrm{d}\epsilon + \alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) \\
=&\int p(\epsilon) \nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} \mathrm{d}\epsilon + \gamma \int p(\epsilon) \mathrm{P}(s'|s, f_\theta(\epsilon; s)) \nabla_\theta V_{\pi_\theta}(s') \mathrm{d}\epsilon \mathrm{d}s' + \alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) \\
=&\int p(\epsilon) \nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} \mathrm{d}\epsilon + \gamma \int p(s \to s', 1, \pi_\theta) \nabla_\theta V_{\pi_\theta}(s') \mathrm{d}s' + \alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)),
\end{aligned}$$

where $p(s \to s', 1, \pi_\theta) = \int p(\epsilon) \mathrm{P}(s'|s, f_\theta(\epsilon; s)) \mathrm{d}\epsilon$.

Now, iterating this formula, we have

$$\begin{aligned}
&\nabla_\theta V_{\pi_\theta}(s) \\
=&\alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) + \int p(\epsilon) \nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} \mathrm{d}\epsilon + \gamma \int p(s \to s', 1, \pi_\theta) \nabla_\theta V_{\pi_\theta}(s') \mathrm{d}s' \\
=&\alpha \nabla_\theta \mathcal{H}(\pi_\theta(\cdot|s)) + \int p(\epsilon) \nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} \mathrm{d}\epsilon + \gamma \int p(s \to s', 1, \pi_\theta) \times
\end{aligned}$$

$$
\begin{aligned}
&\left(\alpha\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s')) + \int p(\epsilon')\nabla_\theta f_\theta(\epsilon';s')\nabla_a Q_{\pi_\theta}(s',a)|_{a=f_\theta(\epsilon';s')}\mathrm{d}\epsilon' + \gamma\int p(s'\to s'',1,\pi_\theta)\nabla_\theta V_{\pi_\theta}(s'')\mathrm{d}s''\right)\mathrm{d}s' \\
=&\alpha\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s)) + \int p(\epsilon)\nabla_\theta f_\theta(\epsilon;s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}\mathrm{d}\epsilon \\
&+\alpha\gamma\int p(s\to s',1,\pi_\theta)\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s'))\mathrm{d}s' + \gamma\int p(s\to s',1,\pi_\theta)p(\epsilon')\nabla_\theta f_\theta(\epsilon';s')\nabla_a Q_{\pi_\theta}(s',a)|_{a=f_\theta(\epsilon';s')}\mathrm{d}\epsilon'\mathrm{d}s' \\
&+\gamma^2\int p(s\to s',1,\pi_\theta)p(s'\to s'',1,\pi_\theta)\nabla_\theta V_{\pi_\theta}(s'')\mathrm{d}s''\mathrm{d}s' \\
=&\alpha\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s)) + \int p(\epsilon)\nabla_\theta f_\theta(\epsilon;s)\nabla_a Q_{\pi_\theta}(s,a)|_{a=f_\theta(\epsilon;s)}\mathrm{d}\epsilon \\
&+\alpha\gamma\int p(s\to s',1,\pi_\theta)\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s'))\mathrm{d}s' + \gamma\int p(s\to s',1,\pi_\theta)p(\epsilon')\nabla_\theta f_\theta(\epsilon';s')\nabla_a Q_{\pi_\theta}(s',a)|_{a=f_\theta(\epsilon';s')}\mathrm{d}\epsilon'\mathrm{d}s' \\
&+\gamma^2\int p(s\to s',2,\pi_\theta)\nabla_\theta V_{\pi_\theta}(s')\mathrm{d}s' \\
=&\dots \\
=&\int\sum_{t=0}^{\infty}\gamma^t p(s\to s',t,\pi_\theta)\left(\alpha\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s')) + \int p(\epsilon')\nabla_\theta f_\theta(\epsilon';s')\nabla_a Q_{\pi_\theta}(s',a)|_{a=f_\theta(\epsilon';s')}\mathrm{d}\epsilon'\right)\mathrm{d}s',
\end{aligned}
$$

where $p(s\to s'',t+1,\pi_\theta) = \int p(s\to s',t,\pi_\theta)p(s'\to s'',1,\pi_\theta)\mathrm{d}s'$.

Furthermore,

$$
\begin{aligned}
&\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s)) \\
=&-\nabla_\theta\int\pi_\theta(a|s)\log\pi_\theta(a|s)\mathrm{d}a \\
=&-\nabla_\theta\int p(\epsilon)\log\pi_\theta(f_\theta(\epsilon;s)|s)\mathrm{d}\epsilon \\
=&-\int p(\epsilon)\nabla_\theta\log\pi_\theta(f_\theta(\epsilon;s)|s)\mathrm{d}\epsilon.
\end{aligned}
$$

Then

$$
\begin{aligned}
&\nabla_\theta V_{\pi_\theta}(s) \\
=&\int\sum_{t=0}^{\infty}\gamma^t p(s\to s',t,\pi_\theta)\left(\alpha\nabla_\theta\mathcal{H}(\pi_\theta(\cdot|s')) + \int p(\epsilon')\nabla_\theta f_\theta(\epsilon';s')\nabla_a Q_{\pi_\theta}(s',a)|_{a=f_\theta(\epsilon';s')}\mathrm{d}\epsilon'\right)\mathrm{d}s' \\
=&\int\sum_{t=0}^{\infty}\gamma^t p(s\to s',t,\pi_\theta)p(\epsilon')\left(\nabla_\theta f_\theta(\epsilon';s')\nabla_a Q_{\pi_\theta}(s',a)|_{a=f_\theta(\epsilon';s')} - \alpha\nabla_\theta\log\pi_\theta(f_\theta(s,\epsilon')|s)\right)\mathrm{d}\epsilon'\mathrm{d}s'.
\end{aligned}
$$

Finally,

$$
\begin{aligned}
&\nabla_\theta J(\theta) \\
=&\nabla_\theta \int p_0(s) V_{\pi_\theta}(s) \mathrm{d}s = \int p_0(s) \nabla_\theta V_{\pi_\theta}(s) \mathrm{d}s \\
=&\int \sum_{t=0}^{\infty} \gamma^t p_0(s) p(s \to s', t, \pi_\theta) p(\epsilon) \left(\nabla_\theta f_\theta(\epsilon; s') \nabla_a Q_{\pi_\theta}(s', a)|_{a=f_\theta(\epsilon;s')} - \alpha \nabla_\theta \log \pi_\theta(f_\theta(\epsilon; s)|s) \right) \mathrm{d}\epsilon \mathrm{d}s' \mathrm{d}s \\
=&\int d^{\pi_\theta, \gamma}(s)\, p(\epsilon) \left(\nabla_\theta f_\theta(\epsilon; s) \nabla_a Q_{\pi_\theta}(s, a)|_{a=f_\theta(\epsilon;s)} - \alpha \nabla_\theta \log \pi_\theta(f_\theta(\epsilon; s)|s) \right) \mathrm{d}\epsilon \mathrm{d}s.
\end{aligned}
$$

□

Note that there is a similar result presented by Haarnoja et al. (2018) for SAC. They obtain it by minimizing the Kullback-Leibler (KL) divergence between the new policy and the policy derived from the exponential of the soft Q-function. Here, we derive it by directly minimizing the objective with the help of the RP gradient. This corollary provides an alternative way to understand SAC.

2.3 Deep Reinforcement Learning

For learning with relatively large-scale MDPs, deep learning (LeCun et al. 2015) are applied into RL. Specifically, neural networks are used to represent policy and value functions. For example, given a policy π, π, V_π, and Q_π can be approximated with neural networks π_θ, V_ϕ, and Q_ψ, respectively, where θ, ϕ, and ψ denote the weights of the neural networks.

In the following, we provide a brief introduction to two important algorithms used in this thesis: deep Q-network (DQN) and proximal policy optimization (PPO).

2.3.1 Deep Q-Network

Based on Q-learning, Mnih et al. (2015) propose the deep Q-network (DQN) algorithm, which uses a neural network (i.e., the Q-network) to approximate the action-value function Q in Q-learning. To mitigate the divergence problem of using function approximators such as neural networks, a target

network is introduced to stabilize training. The target neural network is a copy of the Q-network and is updated periodically for every C_{target} steps, where C_{target} is an integer hyper-parameter. Furthermore, to improve sample efficiency and learning stability, Lin (1992) equipped the learning agent with experience replay, storing recent transitions collected by the agent in a buffer for future use. During a learning update, the agent would sample a mini-batch of transitions and then perform TD learning updates. The full description is listed in Algorithm 1.

Algorithm 1 Deep Q-Network (DQN)

Initialize an experience replay buffer D
Initialize action-value function Q_ψ with random weights ψ
Initialize target action-value function Q_{ψ^-} with random weights $\psi^- = \psi$
Observe initial state s
while the agent is interacting with the environment **do**
    Choose action a for s by ϵ-greedy based on Q
    Take action a, observe r, s'
    Store transition (s, a, r, s') in D and update state $s = s'$
    for every $C_{current}$ steps **do**
        Sample a random mini-batch of transitions $B = \{(s, a, r, s')\}$ from D
        Get update target: $Y(s, a) = r + \gamma \max_{a' \in \mathcal{A}} Q_{\psi^-}(s', a')$ for $(s, a, r, s') \in B$
        Compute DQN loss: $L_{DQN} = \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (Y(s, a) - Q_\psi(s, a))^2$
        Perform a gradient descent step on L_{DQN} with respect to ψ
    Reset $\psi^- = \psi$ for every C_{target} steps

2.3.2 Proximal Policy Optimization

Proximal policy optimization (PPO) is simple to implement, closely related to on-policy learning, and achieves good performance on many tasks (Schulman et al. 2017). First, instead of subtracting the state-value baseline V_{π_θ} as in Corollary 2.1, we use λ-return (Sutton and Barto 2018) G_t^λ to replace $V_{\pi_\theta}(s)$, which has a close relationship to GAE $H_t^{\text{GAE}(\lambda)}$ (Schulman et al. 2016), i.e., $G_t^\lambda = H_t^{\text{GAE}(\lambda)} + V_{\pi_\theta}(s_t)$, where $\lambda \in [0, 1]$. Using λ-returns as baselines significantly reduces the variance of gradient estimations while retaining tolerable gradient biases (Schulman et al. 2016). Next, we constrain the policy update in PPO by using a clipped surrogate objective $\mathbb{E}_{\pi_{\theta_{\text{old}}}}[l_t(\theta)]$, where $l_t(\theta) = \min(\rho_t(\theta)H_t, \text{clip}(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon)H_t)$, where H_t is an estimator of the advantage function, and $\rho_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{\text{old}}}(a_t|s_t)$ is the importance sampling ratio. Then the gradients of

$l_t(\theta)$ can be written as

$$\nabla_\theta l_t(\theta) = \begin{cases} 0, & H_t > 0, \rho_t(\theta) > 1 + \epsilon \\ 0, & H_t < 0, \rho_t(\theta) < 1 - \epsilon \cdot \\ \nabla_\theta \rho_t(\theta) H_t, & \text{otherwise} \end{cases} \tag{2.12}$$

Inspired by Equation (2.12), we can use a modified ratio $\hat{\rho}_t(\theta)$ to simplify,

$$\hat{\rho}_t(\theta) = \begin{cases} 0, & H_t > 0, \rho_t(\theta) > 1 + \epsilon \\ 0, & H_t < 0, \rho_t(\theta) < 1 - \epsilon \cdot \\ \rho_t(\theta), & \text{otherwise} \end{cases} \tag{2.13}$$

Define $l_t(\theta) = \hat{\rho}_t(\theta) H_t$. Easy to verify that this $\nabla_\theta l_t(\theta)$ is the same as Equation (2.12). For a complete description of PPO, please check Algorithm 2.

Algorithm 2 Proximal Policy Optimization (PPO)

Input: initial policy function parameters θ, initial value function parameters ϕ
for $k = 0, 1, 2, \ldots$ **do**
    Collect set of trajectories $\mathcal{D} = \{\tau_i\}$ by running policy π_θ in the environment
    Compute returns G_t
    Compute advantage estimates $H_t = H_t^{\mathrm{GAE}(\lambda)}$
    Normalize H_t using the sample mean and standard deviation of all advantage estimates
    for epoch $= 0, 1, 2, \ldots$ **do**
        Shuffle and slice trajectories $\mathcal{D}$ into mini-batches
        for each mini-batch B **do**
            Set $\hat{\rho}_t(\theta)$ according to Equation (2.13)
            Update policy parameters θ by maximizing the objective: $\mathbb{E}_B[\hat{\rho}_t(\theta) H_t]$
            Update value estimate parameters ϕ by minimizing: $\mathbb{E}_B[(\hat{v}_\phi(S_t) - G_t)^2]$

2.4 Catastrophic Forgetting

When trained on non-independent and non-identically distributed (non-IID) data and optimized with stochastic gradient descent (SGD) algorithms, neural networks tend to forget prior knowledge

abruptly, a phenomenon known as *catastrophic forgetting* (French 1999, McCloskey and Cohen 1989). It is widely observed in continual supervised learning (Hsu et al. 2018, Farquhar and Gal 2018, Van de Ven et al. 2022, Delange et al. 2021) and reinforcement learning (Schwarz et al. 2018, Khetarpal et al. 2022, Atkinson et al. 2021).

2.4.1 Definition

Following Doan et al. (2021), we formally define (task-level) catastrophic forgetting as follows.

Definition 2.1 (Catastrophic Forgetting)**.** *Let $f_{\boldsymbol{w}}(\boldsymbol{x})$ be a neural network with input $\boldsymbol{x}$, parameterized by $\boldsymbol{w}$. This neural network is first trained on task A and then on task B. The resulting weights after training on tasks A and B are denoted as $\boldsymbol{w}_A$ and $\boldsymbol{w}_B$, respectively. Then the catastrophic forgetting of task A after training on task B, with respect to a test dataset D of task A, is defined as*

$$\Delta^{A\to B}(D) = \sum_{(\boldsymbol{x},\boldsymbol{y})\in D} ||f_{\boldsymbol{w}_A}(\boldsymbol{x}) - f_{\boldsymbol{w}_B}(\boldsymbol{x})||_2^2. \tag{2.14}$$

Note that a larger value of $\Delta^{A\to B}$ indicates a greater degree of catastrophic forgetting. When the dataset D contains only one sample, $\Delta^{A\to B}$ reduces to sample-level catastrophic forgetting, which closely relates to Property 2.2 and Property 2.3 in the next section.

2.4.2 Understanding Catastrophic Forgetting via Training Dynamics

Next, we look into the forgetting issue through the lens of neural network training dynamics. To conduct a detailed and fine-grained analysis, we examine and enhance the existing concept of local elasticity (He and Su 2020), proposing several properties that should be achieved or satisfied by continual learning systems.

He and Su (2020) discussed the stability-plasticity dilemma in light of a concept they called local elasticity. A function f is locally elastic if, after being updated with gradient descent at timestep t, $f(\boldsymbol{x})$ is not significantly changed at $\boldsymbol{x}$ that is dissimilar to $\boldsymbol{x}_t$ in a certain sense, and vice versa. For example, we can characterize the dissimilarity with the 2-norm distance. Although He and Su

(2020) show that neural networks with non-linear activation functions are locally elastic in general, the degrees of local elasticity of classical neural networks are not enough to address the forgetting issue empirically, as we will show next. Moreover, there is a lack of theoretical understanding of the connection between network architectures and the degrees of local elasticity.

For simplicity, we consider a regression task. Let a scalar-valued function $f_{\boldsymbol{w}}(\boldsymbol{x})$ be represented as a neural network, parameterized by $\boldsymbol{w}$, with input $\boldsymbol{x}$. $F(\boldsymbol{x})$ is the true function and the loss function is $L(f, F, \boldsymbol{x})$. For example, for squared error, we have $L(f, F, \boldsymbol{x}) = (f_{\boldsymbol{w}}(\boldsymbol{x}) - F(\boldsymbol{x}))^2$. At each time step t, a new sample $\{\boldsymbol{x}_t, F(\boldsymbol{x}_t)\}$ arrives. Given this new sample, to minimize the loss function $L(f, F, \boldsymbol{x}_t)$, we update the weight vector by $\boldsymbol{w}' = \boldsymbol{w} + \Delta_{\boldsymbol{w}}$ where $\Delta_{\boldsymbol{w}}$ is the weight difference. With the stochastic gradient descent (SGD) algorithm, we have $\boldsymbol{w}' = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} L(f, F, \boldsymbol{x}_t)$, where α is the learning rate. So $\Delta_{\boldsymbol{w}} = \boldsymbol{w}' - \boldsymbol{w} = -\alpha \nabla_{\boldsymbol{w}} L(f, F, \boldsymbol{x}_t) = -\alpha \nabla_f L(f, F, \boldsymbol{x}_t) \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t)$. To see how $\Delta_{\boldsymbol{w}}$ relates to the function change for any $\boldsymbol{x}$, we apply Taylor expansion and have

$$
\begin{aligned}
& f_{\boldsymbol{w}'}(\boldsymbol{x}) - f_{\boldsymbol{w}}(\boldsymbol{x}) \\
=& f_{\boldsymbol{w}+\Delta_{\boldsymbol{w}}}(\boldsymbol{x}) - f_{\boldsymbol{w}}(\boldsymbol{x}) \\
=& f_{\boldsymbol{w}}(\boldsymbol{x}) + \langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \Delta_{\boldsymbol{w}} \rangle + O(\Delta_{\boldsymbol{w}}^2) - f_{\boldsymbol{w}}(\boldsymbol{x}) \\
=& \langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \Delta_{\boldsymbol{w}} \rangle + O(\Delta_{\boldsymbol{w}}^2) \\
=& -\alpha \nabla_f L(f, F, \boldsymbol{x}_t) \langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle + O(\Delta_{\boldsymbol{w}}^2),
\end{aligned} \tag{2.15}
$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product or Frobenius inner product depending on the context. In this equation, only the **neural tangent kernel (NTK)** (Jacot et al. 2018, Banerjee et al. 2023) is related to $\boldsymbol{x}$, which is formally defined as

$$
K_{\text{ntk}}(\boldsymbol{x}, \boldsymbol{x}_t; \boldsymbol{w}) = \langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle. \tag{2.16}
$$

This kernel is central to describe the training dynamic of neural networks (Huang and Yau 2020, Belfer et al. 2024), closely related to generalization (Huang et al. 2020, Chen et al. 2020) and forgetting (Riemer et al. 2019, Doan et al. 2021) in deep learning.

Without loss of generality, assume that the original prediction $f_{\boldsymbol{w}}(\boldsymbol{x}_t)$ is wrong, i.e., $f_{\boldsymbol{w}}(\boldsymbol{x}_t) \neq F(\boldsymbol{x}_t)$ and $\nabla_f L(f, F, \boldsymbol{x}_t) \neq 0$. To correct the wrong prediction while avoiding forgetting, after performing a gradient descent step, the function value at $\boldsymbol{x} = \boldsymbol{x}_t$ should be updated and the function values for $\boldsymbol{x} \neq \boldsymbol{x}_t$ should be not be changed. Thus, we expect this NTK to satisfy two properties that are essential for continual learning:

Property 2.1 (Error Correction)**.** *For* $\boldsymbol{x} = \boldsymbol{x}_t$, $\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle \neq 0$.

Property 2.2 (Zero Forgetting)**.** *For* $\boldsymbol{x} \neq \boldsymbol{x}_t$, $\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle = 0$.

In particular, Property 2.1 allows for error correction by optimizing $f_{\boldsymbol{w}'}(\boldsymbol{x}_t)$ towards the true value $F(\boldsymbol{x}_t)$, so that we can learn new knowledge (i.e., update the learned function). We would have $f_{\boldsymbol{w}'}(\boldsymbol{x}) - f_{\boldsymbol{w}}(\boldsymbol{x}) \approx 0$ if $\langle \nabla_{\boldsymbol{w}} f(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle = 0$, failing to correct the wrong prediction at $\boldsymbol{x} = \boldsymbol{x}_t$. Essentially, Property 2.1 requires the gradient norm to be non-zero. On the other hand, Property 2.2 is much harder to be satisfied, especially for nonlinear approximations. To make this property hold, except for $\boldsymbol{x} = \boldsymbol{x}_t$, the neural network f is required to achieve zero forgetting after one step optimization, i.e., $\forall \boldsymbol{x} \neq \boldsymbol{x}_t, f_{\boldsymbol{w}'}(\boldsymbol{x}) = f_{\boldsymbol{w}}(\boldsymbol{x})$. It is the violation of Property 2.2 that leads to the forgetting issue. For tabular cases (e.g., $\boldsymbol{x}$ is a one-hot vector and $f_{\boldsymbol{w}}(\boldsymbol{x})$ is a linear function), this property may hold by sacrificing the generalization ability of deep neural networks. In order to benefit from generalization, we propose Property 2.3 by relaxing Property 2.2:

Property 2.3 (Mild Forgetting)**.** $\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle \approx 0$ *for* $\boldsymbol{x}$ *that is dissimilar to* $\boldsymbol{x}_t$ *in a certain sense.*

The above property is intentionally stated in a loose form to maintain generality. It can be made more precise under specific conditions. For example, consider two positive real numbers ϵ and δ. Then mild forgetting could be that $\forall \boldsymbol{x}$ s.t. $||\boldsymbol{x} - \boldsymbol{x}_t|| \geq \epsilon$, $|\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle| \leq \delta$ holds.

2.5 Knowledge Distillation

Knowledge distillation, introduced by Hinton et al. (2014), is a model compression technique where a compact "student" model is trained to mimic the output of a larger "teacher" network by learning

from the teacher's soft probability outputs, which capture nuanced inter-class relationships. This concept extends earlier work on model compression and teacher-student frameworks (Buciluǎ et al. 2006, Ba and Caruana 2014), which aimed to approximate complex models or ensembles using simpler networks.

While initially applied to convolutional networks for image recognition, knowledge distillation has since been widely adopted in domains, such as natural language processing (Tang et al. 2019, Sanh et al. 2019), speech recognition (Huang et al. 2018), and RL (Rusu et al. 2015, Sun and Fazli 2019, Lai et al. 2020). Variants such as self-distillation (Zhang et al. 2019a), multi-teacher distillation (You et al. 2017), attention transfer (Zagoruyko and Komodakis 2017), and contrastive distillation (Tian et al. 2020) have further extended its applicability. Moreover, it is frequently combined with complementary compression techniques, including pruning (Han et al. 2016), quantization (Hubara et al. 2018), and neural architecture search (Cai et al. 2020a), to achieve highly efficient yet accurate models suitable for deployment on resource-constrained hardware.

2.6 Meta-Gradient Methods

Meta-learning (or learning to learn) (Thrun and Pratt 1998, Vilalta and Drissi 2002, Hospedales et al. 2021, Vettoruzzo et al. 2024) involves two levels of learning and aims not only to perform well on specific tasks but also to acquire meta-knowledge and improve learning algorithms themselves through experience over time. It has been widely applied in machine learning, including tuning hyperparameters (Sutton 1992a, Li et al. 2017), modifying network weights (Schmidhuber 1987), designing improved loss functions (Houthooft et al. 2018, Kirsch et al. 2020, Bechtle et al. 2021), enhancing task representations (Javed and White 2019), and discovering more effective update rules for RL (Oh et al. 2020).

From an optimization perspective, meta-learning can be viewed as a bilevel optimization problem (Zhang et al. 2024). Various optimization techniques have been applied to meta-learning, including stochastic gradient descent (Andrychowicz et al. 2016, Wichrowska et al. 2017, Finn et al. 2017a;b, Franceschi et al. 2018, Bechtle et al. 2021, Liu et al. 2019a), evolutionary algo-

rithms (Schmidhuber 1987, Song et al. 2020, Lu et al. 2022, Houthooft et al. 2018), and RL (Daniel et al. 2016, Duan et al. 2016, Li and Malik 2017).

Among them, we primarily focus on meta-gradient methods (Xu et al. 2018, Sutton 2022), which apply gradient-based optimization to meta-learning. Due to their generality, meta-gradient methods have been widely adopted in deep learning and have demonstrated promising performance in both supervised learning and RL. For instance, they have been used to tune learning rates automatically, thereby accelerating deep neural network training in supervised learning (Jacobs 1988, Sutton 1992a;b, Schraudolph and Sejnowski 1995, Schraudolph 1998; 1999; 2002). In RL, beyond tuning learning rates for improved adaptation (Bagheri et al. 2014, Young et al. 2019), meta-gradient methods have been applied to learn loss functions that significantly enhance learning efficiency (Houthooft et al. 2018, Kirsch et al. 2020, Bechtle et al. 2021), improve exploration strategies based on prior experience (Gupta et al. 2018), adapt dynamic models online (Nagabandi et al. 2019), and discover update rules for RL by interacting with diverse environments (Oh et al. 2020, Kirsch et al. 2022).

Chapter 3

Memory-Efficient Reinforcement Learning with Value-Based Knowledge Consolidation

In reinforcement learning (RL), an agent receives a non-IID stream of experience due to changes in policy, state distribution, the environment dynamics, or simply due to the inherent structure of the environment (Alt et al. 2019). This leads to catastrophic forgetting that previous learning is overridden by later training, resulting in deteriorating performance during single-task training (Ghiassian et al. 2020, Pan et al. 2022a). To mitigate this problem, Lin (1992) equipped the learning agent with experience replay, storing recent transitions collected by the agent in a buffer for future use. By storing and reusing data, the experience replay buffer alleviates the problem of non-IID data and dramatically boosts sample efficiency and learning stability. In deep RL, the experience replay buffer is a standard component, widely applied in value-based algorithms (Mnih et al. 2013; 2015, van Hasselt et al. 2016), policy gradient methods (Schulman et al. 2015; 2017, Haarnoja et al. 2018, Lillicrap et al. 2016b), and model-based methods (Heess et al. 2015, Ha and Schmidhuber 2018, Schrittwieser et al. 2020).

However, using an experience replay buffer is not an ideal solution to catastrophic forgetting.

The memory capacity of the agent is limited by the hardware, while the environment itself may generate an indefinite amount of observations. To store enough information about the environment and get good learning performance, current methods usually require a large replay buffer (e.g., a buffer of a million images). The requirement of a large buffer prevents the application of RL algorithms to the real world since it creates a heavy memory burden, especially for onboard and edge devices (Hayes et al. 2019, Hayes and Kanan 2022). For example, Wang et al. (2023) showed that the performance of SAC (Haarnoja et al. 2018) decreases significantly with limited memory due to hardware constraints of real robots. Smith et al. (2023) demonstrated that a quadruped robot can learn to walk from scratch in 20 minutes in the real-world. However, to be able to train outdoors with enough memory and computation, the authors had to carry a heavy laptop tethered to a legged robot during training, which makes the whole learning process less human-friendly. The world is calling for more memory-efficient RL algorithms.

In this chapter, we propose memory-efficient RL algorithms based on the deep Q-network (DQN) algorithm. Specifically, we assign a new role to the target neural network, which was introduced originally to stabilize training (Mnih et al. 2015). In our algorithms, the target neural network plays the role of a knowledge keeper and helps consolidate knowledge in the action-value network through a consolidation loss. We also introduce a tuning parameter to balance learning new knowledge and remembering past knowledge. With the experiments in both feature-based and image-based environments, we demonstrate that our algorithms, while using an experience replay buffer at least 10 times smaller compared to the experience replay buffer for DQN, still achieve comparable or even better performance.

3.1 Understanding Forgetting from an Objective-Mismatch Perspective

We first use a simple example in supervised learning to shed some light on catastrophic forgetting from an objective-mismatch perspective. Let D be the whole training dataset. We denote the true objective function on D as $L_D(\theta_t)$, where θ_t is a set of parameters at time-step t. Denote B_t as

a subset of D used for training at time-step t. We expect to approximate $L_D(\theta_t)$ with $L_{B_t}(\theta_t)$. For example, B_t could be a mini-batch sampled from D. It could also be a sequence of temporally correlated samples when samples in D come in a stream. When B_t is IID (e.g., sampling B_t uniformly at random from D), there is no objective mismatch since $L_D(\theta_t) = \mathbb{E}_{B_t \sim D}[L_{B_t}(\theta_t)]$ for a specific t. However, when B_t is non-IID, it is likely that $L_D(\theta_t) \neq \mathbb{E}_{B_t \sim D}[L_{B_t}(\theta_t)]$, resulting in the objective mismatch problem. The mismatch between optimizing the true objective and optimizing the objective induced by B_t often leads to catastrophic forgetting. Without incorporating additional techniques, catastrophic forgetting is highly likely when SGD algorithms are used to train neural networks given non-IID data—the optimization objective is simply wrong.

To demonstrate how objective mismatch can lead to catastrophic forgetting, we perform a regression experiment. Specifically, a neural network is trained to approximate a sine function $y = \sin(\pi x)$, where $x \in [0, 2]$. To get non-IID input data, we consider two-stage training. In Stage 1, we generate training samples (x, y), where $x \in [0, 1]$ and $y = \sin(\pi x)$. For Stage 2, $x \in [1, 2]$ and $y = \sin(\pi x)$. The neural network was a multi-layer perceptron with hidden layers $[32, 32]$ and ReLU activation functions. The network is first trained with samples in Stage 1 in the traditional supervised learning style. After that, we continue training the network in Stage 2. We used Adam optimizer with learning rate 0.01. The mini-batch size was 32. For each stage, we trained the network with $1,000$ mini-batch updates. Finally, we plot the learned function after the end of training for each stage.

As shown in Figure 3.1(a), after Stage 1, the learned function fits the true function on $x \in [0, 1]$ almost perfectly. At the end of Stage 2, the learned function also approximates the true function on $x \in [1, 2]$ well. However, the network catastrophically forgets function values it learned on $x \in [0, 1]$. As stated above, due to input distribution shift, the objective function we minimize is defined by training samples only from one stage (i.e., $x \in [0, 1]$ or $x \in [1, 2]$) that are not enough to reconstruct the true objective properly which is defined on the whole training set (i.e., $x \in [0, 2]$). Much of the previously learned knowledge is lost while optimizing a wrong objective.

A similar phenomenon also exists in single RL tasks. Specifically, we show that, without a large replay buffer, DQN can easily forget the optimal action after it has learned it. We test DQN

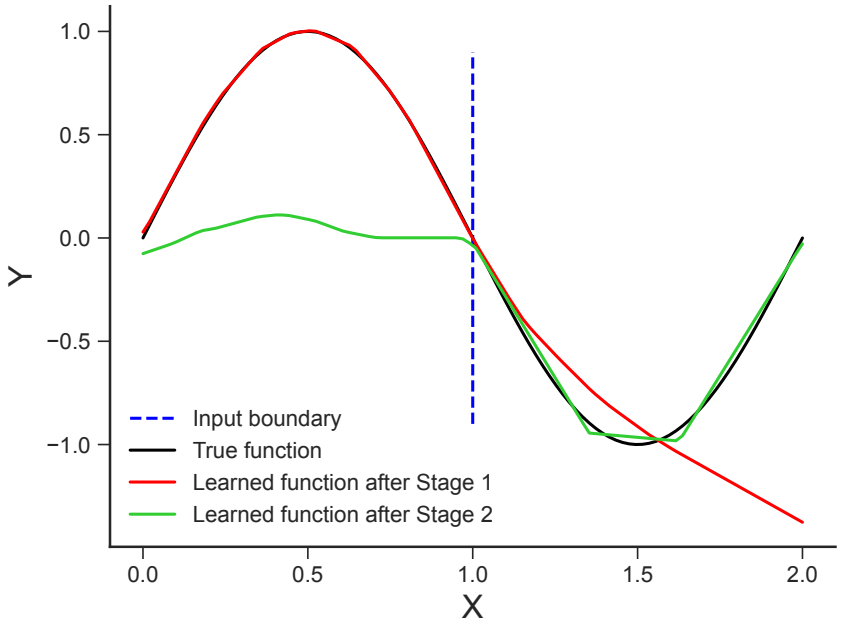


Figure 3.1: A visualization of learned functions trained with SGD. In Stage 1, we generate training samples (x, y), where $x \in [0, 1]$ and $y = \sin(\pi x)$. In Stage 2, $x \in [1, 2]$ and $y = \sin(\pi x)$. The blue dotted line shows the boundary of the input space for Stage 1 and 2. Clearly, after Stage 2, the network catastrophically forgets function values it learned in Stage 1.

in Mountain Car (Sutton and Barto 2018), with and without using a large replay buffer. Denote *DQN(S)* as DQN using a tiny (i.e., 32) experience replay buffer. We first randomly sample a state S and then record its greedy action (i.e., $\arg\max_a Q(S, a)$) through the whole training process, as shown in Figure 3.2.[1] Note that the optimal action for this state is 1. We perform many runs to verify the result and only show the single-run result here for better visualization. We observe that, when using a large buffer (i.e., $10K$), DQN does not suffer much from forgetting. However, when using a small buffer (i.e., 32), DQN(S) keeps forgetting and relearning the optimal action, demonstrating the forgetting issue in single RL tasks. MeDQN(U) is our algorithm which will be introduced later. As shown in Figure 3.2, MeDQN(U) consistently chooses the optimal action 1 as its greedy action and suffers much less from forgetting, even though it also uses a tiny replay buffer with size 32.

[1] To be specific, the randomly sampled state is $S = [-0.70167243, 0.04185214]$.

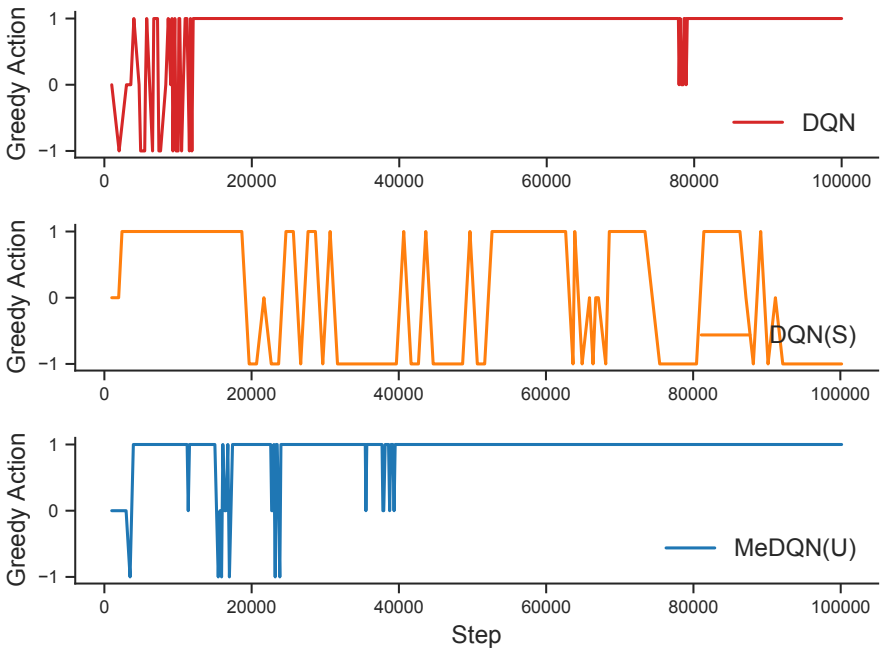


Figure 3.2: The greedy actions of a randomly sampled state for different methods during training in Mountain Car. There are 3 actions in total, and the optimal action is 1. Without a large replay buffer, DQN(S) keeps forgetting and relearning the optimal action, while DQN and MeDQN(U) suffer much less from forgetting.

3.2 Related Work

The key to reducing catastrophic forgetting is to preserve past acquired knowledge while acquiring new knowledge. In this section, we will first discuss existing methods for continual supervised learning and then methods for continual RL. Since we can not list all related methods here, we encourage readers to check recent surveys (Khetarpal et al. 2022, Delange et al. 2021).

3.2.1 Supervised Learning

In the absence of memory constraints, rehearsal methods, also known as *replay*, are usually considered one of the most effective methods in continual supervised learning (Kemker et al. 2018, Farquhar and Gal 2018, Van de Ven et al. 2022, Delange et al. 2021). Concretely, these methods retain knowledge explicitly by storing previous training samples (Rebuffi et al. 2017, Riemer et al.

2018, Hayes et al. 2019, Aljundi et al. 2019a, Chaudhry et al. 2019, Jin et al. 2020). In generative replay methods, samples are stored in generative models rather than in a buffer (Shin et al. 2017, Kamra et al. 2017, Van de Ven and Tolias 2018, Ramapuram et al. 2020, Choi et al. 2021). These methods exploit a dual memory system consisting of a student and a teacher network. The current training samples from a data buffer are first combined with pseudo samples generated from the teacher network and then used to train the student network with knowledge distillation (Hinton et al. 2014). Some previous methods carefully select and assign a subset of weights in a large network to each task (Mallya and Lazebnik 2018, Sokar et al. 2021, Fernando et al. 2017, Serra et al. 2018, Masana et al. 2021, Li et al. 2019, Yoon et al. 2018) or assign a task-specific network to each task (Rusu et al. 2016, Aljundi et al. 2017). Changing the update rule is another approach to reducing forgetting. Methods such as GEM (Lopez-Paz and Ranzato 2017), OWM (Zeng et al. 2019), and ODG (Farajtabar et al. 2020) protect obtained knowledge by projecting the current gradient vector onto some constructed space related to previous tasks. Finally, parameter regularization methods (Kirkpatrick et al. 2017, Schwarz et al. 2018, Zenke et al. 2017, Aljundi et al. 2019b) reduce forgetting by encouraging parameters to stay close to their original values with a regularization term so that more important parameters are updated more slowly.

3.2.2 Reinforcement Learning

RL tasks are natural playgrounds for continual learning research since the input is a stream of temporally structured transitions (Khetarpal et al. 2022). In continual RL, most works focus on incremental task learning where tasks arrive sequentially with clear task boundaries. For this setting, many methods from continual supervised learning can be directly applied, such as EWC (Kirkpatrick et al. 2017) and MER (Riemer et al. 2018). Many works also exploit similar ideas from continual supervised learning to reduce forgetting in RL. For example, Ammar et al. (2014) and Mendez et al. (2020) assign task-specific parameters to each task while sharing a reusable knowledge base among all tasks. Mendez et al. (2022) and Isele and Cosgun (2018) use an experience replay buffer to store transitions of previous tasks for knowledge retention. The above methods cannot be applied to our case directly since they require either clear task boundaries or large buffers. The student-teacher

dual memory system is also exploited in multi-task RL, inducing behavioral cloning by function regularization between the current network (the student network) and its old version (the teacher network) (Rolnick et al. 2019, Kaplanis et al. 2019, Atkinson et al. 2021). Note that in continual supervised learning, there are also methods that regularize a function directly (Shin et al. 2017, Kamra et al. 2017, Titsias et al. 2020). Our work is inspired by this approach in which the target Q neural network plays the role of the teacher network that regularizes the student network (i.e., current Q neural network) directly.

In single RL tasks, the forgetting issue is under-explored and unaddressed, as the issue is masked by using a large replay buffer. In this work, we aim to develop memory-efficient single-task RL algorithms while achieving high sample efficiency and training performance by reducing forgetting.

3.3 MeDQN: Memory-Efficient Deep Q-Network

In this section, we present our method. We begin by introducing knowledge consolidation, which forms the foundation of our approach. We then introduce two variants of the proposed method.

3.3.1 Knowledge Consolidation

Originally, Hinton et al. (2014) proposed distillation to transfer knowledge between different neural networks effectively. In this thesis, we refer to *knowledge consolidation* as a special case of distillation that transfers information from an old copy of this network (e.g., the target network, parameterized by θ^-) to the network itself (e.g., the current network, parameterized by θ), consolidating the knowledge that is already contained in the network. Unlike methods like EWC (Kirkpatrick et al. 2017) and SI (Zenke et al. 2017) that regularize parameters, knowledge consolidation regularizes the function directly. Formally, we define the (vanilla) consolidation loss as follows:

$$L^V_{consolid}(\theta) = \mathbb{E}_{(S,A)\sim p(\cdot,\cdot)}\left[\left(Q(S,A;\theta) - Q(S,A;\theta^-)\right)^2\right],$$

where $p(s,a)$ is a sampling distribution over the state-action pair s,a. To retain knowledge, the state-action space should be covered by $p(s,a)$ sufficiently, such as $p(s,a) = d^\pi(s)\pi(a|s)$ or $p(s,a) = d^\pi(s)\mu(a)$, where π is the ϵ-greedy policy, d^π is the stationary state distribution of π, and μ is a uniform distribution over $\mathcal{A}$. Our preliminary experiment shows that $p(s,a) = d^\pi(s)\mu(a)$ is a better choice compared with $p(s,a) = d^\pi(s)\pi(a|s)$; so we use $p(s,a) = d^\pi(s)\mu(a)$ in this work. To summarize, the following consolidation loss is used in this work:

$$L_{consolid}(\theta) = \mathbb{E}_{S\sim d^\pi}\left[\sum_{A\in\mathcal{A}}\left(Q(S,A;\theta) - Q(S,A;\theta^-)\right)^2\right], \tag{3.1}$$

Intuitively, minimizing the consolidation loss can preserve previously learned knowledge by penalizing $Q(s,a;\theta)$ for deviating from $Q(s,a;\theta^-)$ too much. In general, we may also use other loss functions, such as the Kullback–Leibler (KL) divergence $D_{\mathrm{KL}}(\pi(\cdot|s)||\hat{\pi}(\cdot|s))$, where π and $\hat{\pi}$ can be softmax policies induced by action values, that is, $\pi(a|s) \propto \exp(Q(s,a;\theta))$ and $\hat{\pi}(a|s) \propto \exp(Q(s,a;\theta^-))$. For simplicity, we use the mean squared error loss, which also proves to be effective, as shown in our experiments.

Given a mini-batch B consisting of transitions $\tau = (s,a,r,s')$, the DQN loss is defined as

$$L_{DQN}(\theta) = \frac{1}{|B|}\sum_{\tau\in B}\left(r + \gamma\max_{a'}Q(s',a';\theta^-) - Q(s,a;\theta)\right)^2.$$

We combine the two losses to obtain the final training loss for our algorithm

$$L = L_{DQN} + \lambda L_{consolid},$$

where λ is a positive scalar.

Note that L_{DQN} helps the Q network learn **new** knowledge by correcting wrong predictions of Q. In contrast, $L_{consolid}$ is used to preserve **old** knowledge by consolidating information from the target network to the current network. By combining them with a weighting parameter λ, we balance learning and preserving knowledge simultaneously. Moreover, since $L_{consolid}$ acts as a functional regularizer, the parameter θ may change significantly as long as the function values of

Algorithm 3 Memory-Efficient DQN with Uniform State Sampling (MeDQN(U))

Initialize a small experience replay buffer D
Initialize state lower bound s_{LOW} to $[\infty, \cdots, \infty]$ and upper bound s_{HIGH} to $[-\infty, \cdots, -\infty]$
Initialize action-value function with random weights θ
Initialize target action-value function with random weights $\theta^- = \theta$
Observe initial state s
while agent is interacting with the environment **do**
    Update state bounds: $s_{LOW} = \min(s_{LOW}, s)$, $s_{HIGH} = \max(s_{HIGH}, s)$
    Take action a chosen by ϵ-greedy based on Q, observe r, s'
    Store transition (s, a, r, s') in D and update state $s = s'$
    for every $C_{current}$ steps **do**
        Get all transitions $B = \{(s, a, r, s')\}$ in D
        for $i = 1$ to E **do**
            Compute DQN loss L_{DQN}
            Sample a random mini-batch states $B^{state} = \{s\}$ uniformly from $[s_{LOW}, s_{HIGH}]$
            Compute consolidation loss $L^U_{consolid}$ given B^{state}
            Compute the final training loss: $L = L_{DQN} + \lambda L^U_{consolid}$
            Perform a gradient descent step on L with respect to θ
        Reset $\theta^- = \theta$ for every C_{target} steps

the current network remain close to the target network.

There is still one problem left: how to get $d^\pi(s)$? In general, it is hard to compute the exact form of $d^\pi(s)$. Instead, we use random sampling, as we will show next.

3.3.2 Uniform State Sampling

One of the simplest ways to approximate $d^\pi(s)$ is with a uniform distribution on $\mathcal{S}$. Formally, we define this version of consolidation loss as

$$L^U_{consolid}(\theta) = \mathbb{E}_{S\sim U}\left[\sum_{A\in\mathcal{A}} \left(Q(S, A; \theta) - Q(S, A; \theta^-)\right)^2\right],$$

where U is a uniform distribution over the state space $\mathcal{S}$.

The intuition behind uniform state sampling is that sometimes we do not necessarily need on-policy states (i.e., states sampled from d^π) to achieve good knowledge consolidation. Although randomly uniformly generated states may not be meaningful, they are enough to induce good knowledge consolidation (i.e., $L_{consolid} \approx 0$) in some cases. For example, considering linear function

approximations that $Q(s,a;\theta) = x^\top\theta$ and $Q(s,a;\theta^-) = x^\top\theta^-$, where $\theta \in \mathbb{R}^n$, $\theta^- \in \mathbb{R}^n$, and $x = (s,a) \in \mathbb{R}^n$. To achieve perfect knowledge consolidation ($L_{consolid} = 0$), it is required to find $\theta = \theta^-$ by minimizing $L_{consolid}$. Let $\{x_1, x_2, \ldots, x_n\}$ be n points randomly uniformly sampled from $\mathcal{S} \times \mathcal{A}$ and set $y_i = x_i^\top\theta^-$ for every i. Denote $X = [x_1; \cdots; x_n]$ and $Y = [y_1, \cdots, y_n]^\top$. The problem can be reformulated as finding θ such that $X^\top\theta = Y$. It is known that the random matrix X is full rank with a high probability (Cooper 2000). In this case, we can get the optimal θ with $\theta = (X^\top)^{-1}Y$ and thus achieve perfect knowledge consolidation ($L_{consolid} = 0$). Note that $\{x_1, x_2, \ldots, x_n\}$ are all randomly generated; they are not necessary to be state-action pairs sampled from real trajectories. Note that perfect knowledge consolidation ($L_{consolid} = 0$) is not preferred in practice, since we still need to update the Q function during training for better policy evaluation. Instead, we want to achieve good knowledge consolidation by keeping $L_{consolid}$ close to zero.

Theoretically, assuming that the size of the state space $|\mathcal{S}|$ is finite, we have $\Pr(S = s) = 1/|\mathcal{S}|$ for any $s \in \mathcal{S}$. Together with $d^\pi(s) \leq 1$, we then have

$$\begin{aligned} L_{consolid}(\theta) &= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \big(Q(s,a;\theta) - Q(s,a;\theta^-)\big)^2 \\ &\leq \sum_{s\in\mathcal{S}} \sum_{a\in\mathcal{A}} \big(Q(s,a;\theta) - Q(s,a;\theta^-)\big)^2 \\ &= |\mathcal{S}| L^U_{consolid}(\theta). \end{aligned} \tag{3.2}$$

Essentially, minimizing $|\mathcal{S}|L^U_{consolid}$ minimizes an upper bound of $L_{consolid}$. As long as $L^U_{consolid}$ is small enough, we can achieve good knowledge consolidation with a low consolidation loss $L_{consolid}$. In the extreme case, $L^U_{consolid} = 0$ leads to $L_{consolid} = 0$.

In practice, we may not know $\mathcal{S}$ in advance. To solve this problem, we maintain state bounds s_{LOW} and s_{HIGH} as the lower and upper bounds of all observed states, respectively. Note that both s_{LOW} and s_{HIGH} are two state vectors with the same dimension as a state in $\mathcal{S}$. Assume $\mathcal{S} \subseteq \mathbb{R}^n$. Initially, we set $s_{LOW} = [\infty, \cdots, \infty] \in \mathbb{R}^n$ and $s_{HIGH} = [-\infty, \cdots, -\infty] \in \mathbb{R}^n$. For each newly

received $s \in \mathbb{R}^n$, we update state bounds with

$$s_{LOW} = \min(s_{LOW}, s) \text{ and } s_{HIGH} = \max(s_{HIGH}, s).$$

Here, both min and max are element-wise operations. During training, we sample pseudo-states uniformly from the interval $[s_{LOW}, s_{HIGH}]$ to help compute the consolidation loss.

We name our algorithm that uses uniform state sampling as memory-efficient DQN with uniform state sampling, denoted as *MeDQN(U)* and shown in Algorithm 3. Compared with DQN, MeDQN(U) has several changes. First, the experience replay buffer D is tiny (Line 1). In practice, we set the buffer size to the mini-batch size to apply mini-batch gradient descent. Second, we maintain state bounds and update them at every step (Line 7). Moreover, to extract as much information from a small replay buffer, we use the same data to train the Q function for E times (Line 13–19). In practice, we find that a small E (e.g.,1–4) is enough to perform well. Finally, we apply knowledge consolidation by adding a consolidation loss to the DQN loss as the final training loss (Line 16–17).

Algorithm 4 Memory-Efficient DQN with Real State Sampling (MeDQN(R))

1: Initialize an experience replay buffer D
2: Initialize the current action-value function with random weights θ
3: Initialize the target action-value function with random weights $\theta^- = \theta$
4: Observe initial state s
5: **while** agent is interacting with the environment **do**
6: Choose action a by ϵ-greedy based on Q
7: Take action a, observe r, s'
8: Store transition (s, a, r, s') in D and update state $s = s'$
9: **for** every $C_{current}$ steps **do**
10: Sample a random mini-batch of transitions $B = \{(s, a, r, s')\}$ from D
11: **for** $i = 1$ to E **do**
12: Compute DQN loss L_{DQN}
13: Sample a random mini-batch of states $B^{state} = \{s\}$ from D
14: Compute consolidation loss $L^R_{consolid}$ given B^{state}
15: Compute the final training loss: $L = L_{DQN} + \lambda L^R_{consolid}$
16: Perform a gradient descent step on L with respect to θ
17: Reset $\theta^- = \theta$ for every C_{target} steps

3.3.3 Real State Sampling

When the state space $\mathcal{S}$ is super large, the agent is unlikely to visit every state in $\mathcal{S}$. Thus, for a policy π, the visited state set $\mathcal{S}^\pi := \{s \in \mathcal{S} | d^\pi(s) > 0\}$ is expected to be a very small subset of $\mathcal{S}$. In this case, a uniform distribution over $\mathcal{S}$ is far from a good estimation of d^π. A small number of states (e.g., one mini-batch) generated from uniform state sampling cannot cover $\mathcal{S}^\pi$ well enough, resulting in poor knowledge consolidation for the Q function. In other words, we may still forget previously learned knowledge (i.e., the action values over $\mathcal{S}^\pi \times \mathcal{A}$) catastrophically. This intuition can also be understood from the view of upper bound minimization. In Equation (3.2), as the upper bound of $L_{consolid}$, $|\mathcal{S}|L^U_{consolid}$ is large when $\mathcal{S}$ is large. Even if $L^U_{consolid}$ is minimized to a small value, the upper bound $|\mathcal{S}|L^U_{consolid}$ may still be too large, leading to poor consolidation.

To overcome the shortcoming of uniform state sampling, we propose real state sampling. Specifically, previously observed states are stored in a state replay buffer D_s and real states are sampled from D_s for knowledge consolidation. Compared with uniform state sampling, states sampled from a state replay buffer have a larger overlap with $\mathcal{S}^\pi$, acting as a better approximation of d^π. Formally, we define the consolidation loss using real state sampling as

$$L^R_{consolid}(\theta) = \mathbb{E}_{S \sim D_s}\left[\sum_{A \in \mathcal{A}} \left(Q(S, A; \theta) - Q(S, A; \theta^-)\right)^2\right].$$

In practice, we sample states from the experience replay buffer D. We name our algorithm that uses real state sampling as memory-efficient DQN with real state sampling, denoted as *MeDQN(R)*. The algorithm description is shown in Algorithm 4. Similar to MeDQN(U), we also use the same data to train the Q function for E times and apply knowledge consolidation by adding a consolidation loss. The main difference is that the experience replay buffer used in MeDQN(R) is relatively large while the experience replay buffer in MeDQN(U) is tiny (i.e., one mini-batch size). However, as we will show next, the experience replay buffer used in MeDQN(R) can still be significantly smaller than the experience replay buffer used in DQN.

3.4 Experiments

In this section, we first verify that knowledge consolidation helps mitigate the objective mismatch problem and reduce forgetting. Next, we propose a strategy to better balance learning and remembering. We then demonstrate that our algorithms achieve comparable or superior performance to DQN in both low-dimensional and high-dimensional tasks. Moreover, we verify through an ablation study that knowledge consolidation is key to achieving both memory efficiency and high performance. Finally, we show that knowledge consolidation also enhances the robustness of our algorithms to varying buffer sizes.

3.4.1 The Effectiveness of Knowledge Consolidation

To show that knowledge consolidation helps mitigate the objective mismatch problem, we first apply it to solve the task of approximating $\sin(\pi x)$, as presented in Section 3.1. Denote the neural network and the target neural network as $f(x;\theta)$ and $f(x;\theta^-)$, respectively. The true loss function is defined as $L_{true} = \mathbb{E}_{x\in[0,2]}[(f(x;\theta) - y)^2]$, where $y = \sin(\pi x)$. We first train the network $f(x;\theta)$ in Stage 1 (i.e., $x \in [0,1]$) without knowledge consolidation. We also maintain input bounds $[x_{LOW}, x_{HIGH}]$, similar to Algorithm 3. At the end of Stage 1, we set $\theta^- = \theta$. At this moment, both $f(x;\theta)$ and $f(x;\theta^-)$ are good approximations of the true function for $x \in [0,1]$; and $[x_{LOW}, x_{HIGH}] \approx [0,1]$. Next, we continue to train $f(x;\theta)$ with $x \in [1,2]$. To apply knowledge consolidation, we sample $\hat{x}$ from input bounds $[x_{LOW}, x_{HIGH}]$ uniformly. Finally, we add the consolidation loss to the training loss and get the following:

$$L(\theta) = \mathbb{E}_{x\in[1,2]}[(f(x;\theta) - y)^2] + \mathbb{E}_{\hat{x}\in[x_{LOW},x_{HIGH}]}[(f(\hat{x};\theta) - f(\hat{x};\theta^-))^2].$$

By adding the consolidation loss, L becomes a good approximation of the true loss function, which helps preserve the knowledge learned in Stage 1 while learning new knowledge in Stage 2 (Figure 3.3). Note that in the whole training process, we do not save previously observed samples (x, y) explicitly. The knowledge of Stage 1 is stored in the target model $f(x;\theta^-)$ and then consolidated to $f(x;\theta)$.

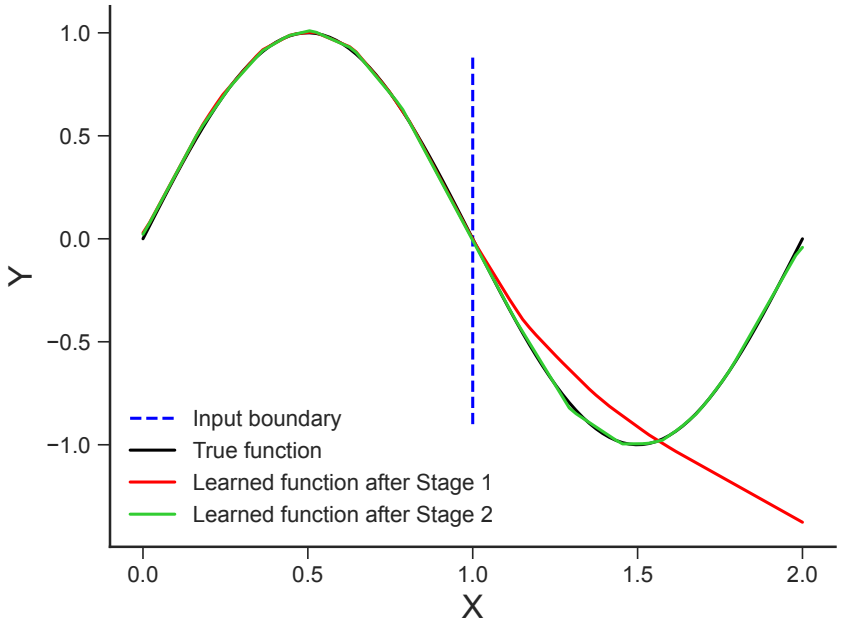


Figure 3.3: A visualization of learned functions trained with SGD and knowledge consoidation. In Stage 1, we generate training samples (x, y), where $x \in [0, 1]$ and $y = \sin(\pi x)$. In Stage 2, $x \in [1, 2]$ and $y = \sin(\pi x)$. The blue dotted line shows the boundary of the input space for Stage 1 and 2. It shows that knowledge consoidation helps preserve the knowledge learned in Stage 1 while learning new knowledge in Stage 2.

Next, we show that combined with knowledge consolidation, MeDQN(U) is able to reduce the forgetting issue in single RL tasks even with a tiny replay buffer. We repeat the RL training process in Section 3.1 for MeDQN(U) and record its greedy action during training. As shown in Figure 3.2, while DQN(S) keeps forgetting and relearning the optimal action, both DQN and MeDQN(U) are able to consistently choose the optimal action 1 as its greedy action, suffering much less for forgetting. Note that both MeDQN(U) and DQN(S) use a tiny replay buffer with size 32 while DQN uses a much larger replay buffer with size $10K$. This experiment demonstrates the forgetting issue in single RL tasks and the effectiveness of our method in mitigating forgetting. More training details in Mountain Car can be found in Section 3.4.3.

We repeat the RL training process in Section 3.1 for MeDQN(U) and record its greedy action during training. As shown in Figure 3.2, while DQN(S) keeps forgetting and relearning the optimal

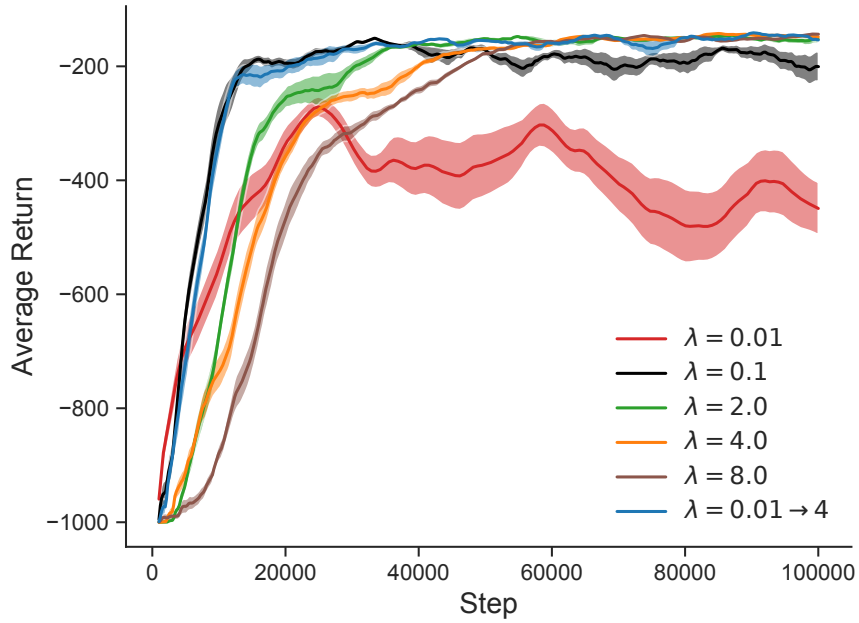


Figure 3.4: A comparison of different strategies to balance learning and preservation for MeDQN(U) in MountainCar-v0. When λ is small, MeDQN(U) learns quickly initially but becomes slower and unstable over time, leading to performance degradation. Increasing λ slows early learning but improves training stability and final performance. All results are averaged over 20 runs, with the shaded area representing two standard errors.

action, both DQN and MeDQN(U) are able to consistently choose the optimal action 1 as its greedy action, suffering much less for forgetting. Note that both MeDQN(U) and DQN(S) use a tiny replay buffer with size 32 while DQN uses a much larger replay buffer with size $10K$. This experiment demonstrates the forgetting issue in single RL tasks and the effectiveness of our method in mitigating forgetting. More training details in Mountain Car can be found in Section 3.4.3.

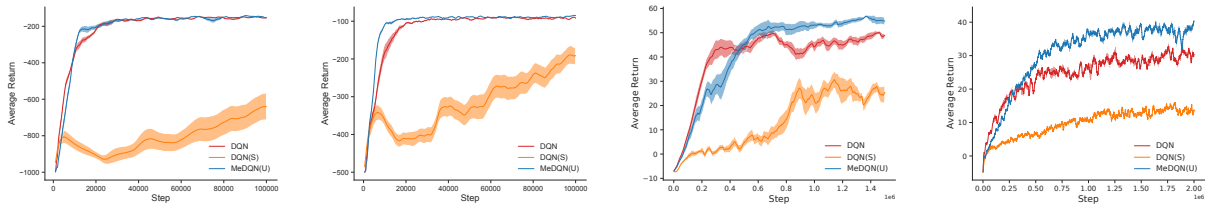
3.4.2 Balancing Learning and Remembering

In Section 3.3.1, we claimed that λ could balance between learning new knowledge and preserving old knowledge. In this section, we used Mountain Car (Sutton and Barto 2018) as a testbed to verify this claim. Specifically, a fixed λ is chosen from $\{0.01, 0.1, 2, 4, 8\}$. The mini-batch size is 32. The experience replay buffer size in MeDQN(U) is 32. The update epoch $E = 4$, the target network

update frequency $C_{target} = 100$, and the current network update frequency $C_{current} = 1$. We chose learning rate from $\{1e-2, 3e-3, 1e-3, 3e-4, 1e-4\}$ and reported the best results averaged over 20 runs for different λ, as shown in Figure 3.4.

When λ is small (e.g., $\lambda = 0.1$ or $\lambda = 0.01$), giving a small weight to the consolidation loss, MeDQN(U) learns fast at the beginning. However, as training continues, the learning becomes slower and unstable; the performance drops. As we increase λ, although the initial learning is getting slower, the training process becomes more stable, resulting in higher performance. These phenomena align with our intuition. At first, not much knowledge is available for consolidation; learning new knowledge is more important. A small λ lowers the weight of $L_{consolid}$ in the training loss, thus speeding up learning initially. As training continues, more knowledge is learned, and knowledge preservation is vital to performance. A small λ fails to protect old knowledge, while a larger λ helps consolidate knowledge more effectively, stabilizing the learning process.

Inspired by these results, we propose a new strategy to balance learning and preservation. Specifically, λ is no longer fixed but linearly increased from a small value λ_{start} to a large value λ_{end}. This mechanism encourages knowledge learning at the beginning and information retention in later training. We increased λ from 0.01 to 4 linearly with respect to the training steps for this experiment. In this setting, we observed that learning is fast initially, and then the performance stays at a high level stably towards the end of training. Given the success of this linearly increasing strategy, we applied it in all of the following experiments for MeDQN.



(a) MountainCar-v0 (b) Acrobot-v1 (c) Catcher (d) Pixelcopter

Figure 3.5: Evaluation in low-dimensional tasks. The results for MountainCar-v0 and Acrobot-v1 are averaged over 20 runs. The results for Catcher and Pixelcopter are averaged over 10 runs. The shaded areas represent two standard errors. MeDQN(U) outperforms DQN in all four tasks, even though it uses an experience replay buffer with one mini-batch size.

3.4.3 Evaluation in Low-Dimensional Tasks

We choose four tasks with low-dimensional inputs from Gym (Brockman et al. 2016) and PyGame Learning Environment (Tasfi 2016): MountainCar-v0 (2), Acrobot-v1 (6), Catcher (4), and Pixelcopter (7), where numbers in parentheses are input state dimensions.

For DQN, we use the same hyper-parameters and training settings as in Lan et al. (2020). Specifically, for MountainCar-v0 and Acrobot-v1, the neural network is a multi-layer perceptron with hidden layers $[32, 32]$; the best learning rate is selected from $\{1e-2, 3e-3, 1e-3, 3e-4, 1e-4\}$ with grid search; Adam is used to optimize network parameters; all algorithms are trained for $1e5$ steps. For Catcher and Pixelcopter, the neural network is a multi-layer perceptron with hidden layers $[64, 64]$; the best learning rate is selected from $\{1e-3, 3e-4, 1e-4, 3e-5, 1e-5\}$ with grid search; RMSprop is used to optimize network parameters. In Catcher, algorithms are trained for $1.5e6$ steps; in Pixelcopter, algorithms are trained for $2e6$ steps. The discount factor is 0.99. The mini-batch size is 32. The buffer size for DQN is $10,000$ in all tasks. For first $1,000$ exploration steps, we only collected transitions without learning. ϵ-greedy is applied as the exploration strategy with ϵ decreasing linearly from 1.0 to 0.01 in $1,000$ steps. After $1,000$ steps, ϵ is fixed to 0.01.

For MeDQN, $\lambda_{start} = 0.01$; λ_{end} is chosen from $\{1, 2, 4\}$; E is selected from $\{1, 2, 4\}$. We choose $C_{current}$ in $\{1, 2, 4, 8\}$. Moreover, we include *DQN(S)* as another baseline, which uses a tiny (i.e., 32) replay buffer. The buffer size is the only difference between DQN and DQN(S). The experience replay buffer size for MeDQN(U) is 32. We set $\lambda_{start} = 0.01$ for MeDQN(U) and tune learning rate for all algorithms. Except for tuning the update epoch E, the current network update frequency $C_{current}$, and λ_{end}, we use the same hyper-parameters and training settings as DQN for MeDQN(U). Other hyper-parameter choices are presented in Table 3.1.

For MountainCar-v0 and Acrobot-v1, we report the best results averaged over 20 runs. For Catcher and Pixelcopter, we report the best results averaged over 10 runs. The learning curves of the best results for all algorithms are shown in Figure 3.5, where the shaded area represents two standard errors. All curves are smoothed using an exponential average. As we can see from Figure 3.5, when using a small buffer, DQN(S) has higher instability and lower performance than

Table 3.1: The hyper-parameters of different algorithms for tasks in Figure 3.5.

(a) MountainCar-v0

Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**
learning rate	1e-2	1e-3	1e-3
experience replay buffer size	1e4	32	32
E	/	/	4
λ_{end}	/	/	4
C_{target}	100	100	100
$C_{current}$	8	1	1

(b) Acrobot-v1

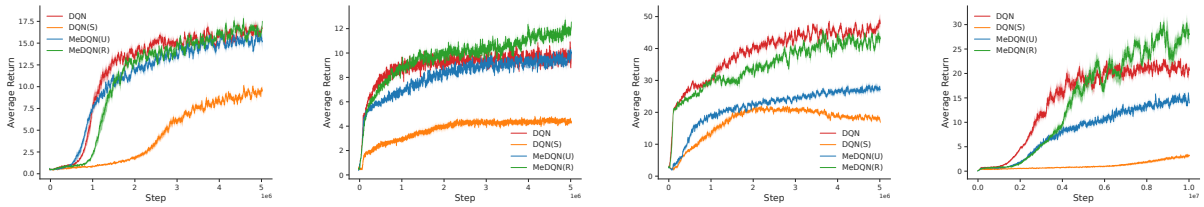
Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**
learning rate	1e-3	3e-4	3e-4
experience replay buffer size	1e4	32	32
E	/	/	1
λ_{end}	/	/	2
C_{target}	100	100	100
$C_{current}$	1	1	1

(c) Catcher

Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**
learning rate	1e-4	1e-5	3e-5
experience replay buffer size	1e4	32	32
E	/	/	4
λ_{end}	/	/	4
C_{target}	200	200	200
$C_{current}$	1	1	2

(d) Pixelcopter

Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**
learning rate	3e-5	1e-5	3e-4
experience replay buffer size	1e4	32	32
E	/	/	1
λ_{end}	/	/	1
C_{target}	200	200	200
$C_{current}$	1	1	8



(a) Asterix (MinAtar) (b) Breakout (MinAtar) (c) Space Invaders (MinAtar) (d) Seaquest (MinAtar)

Figure 3.6: Evaluation in high-dimensional MinAtar tasks. All results are averaged over 10 runs, with the shaded areas representing two standard errors. MeDQN(R) is comparable with DQN even though it uses a much smaller experience replay buffer (10% of the replay buffer size in DQN).

DQN, which uses a large buffer. MeDQN(U) outperforms DQN in all four games. The results are inspiring, given that MeDQN(U) only uses a tiny experience replay buffer. We conclude that MeDQN(U) is much more memory-efficient while achieving similar or better performance compared to DQN in low-dimensional tasks. Given that MeDQN(U) already performs well while being very memory-efficient, we did not further test MeDQN(R) in these low-dimensional tasks.

3.4.4 Evaluation in High-dimensional Tasks

To further evaluate our algorithms, we choose four tasks with high-dimensional image inputs from MinAtar (Young and Tian 2019): Asterix ($10\times10\times4$), Seaquest ($10\times10\times10$), Breakout ($10\times10\times4$), and Space Invaders ($10 \times 10 \times 6$), where numbers in parentheses are input dimensions.

For DQN, we reuse the hyper-parameters and neural network setting in Young and Tian (2019). The discount factor is 0.99. The mini-batch size is 32. In Seaquest (MinAtar), all algorithms are trained for $1e7$ steps. Except that, all algorithms are trained for $5e6$ steps in other tasks. For the first $5,000$ exploration steps, we only collect transitions without learning. ϵ-greedy is applied as the exploration strategy with ϵ decreasing linearly from 1.0 to 0.1 in $5,000$ steps. After $5,000$ steps, ϵ was fixed to 0.1. Following Young and Tian (2019), we used smooth L1 loss in PyTorch and centered RMSprop optimizer with $\alpha = 0.95$ and $\epsilon = 0.01$. The best learning rate was chosen from $\{3e-3, 1e-3, 3e-4, 1e-4, 3e-5\}$ with grid search. We also reused the settings of neural networks. For both MeDQN(R) and MeDQN(U), $\lambda_{start} = 0.01$; λ_{end} was chosen from $\{2, 4\}$; E was selected from $\{1, 2\}$. We chose $C_{current}$ in $\{4, 8, 16, 32\}$.

For DQN, the buffer size is $100,000$ and the target network update frequency $C_{target} = 1,000$. For MeDQN(R), $C_{target} = 1,000$ and the buffer size is 10% of the buffer size in DQN, i.e., $10,000$. The replay buffer size for MeDQN(U) is 32. For MeDQN(U), a smaller C_{target} is better and we set $C_{target} = 300$. Other hyper-parameter choices are presented in Table 3.2.

The learning curves of best results are shown in Figure 3.6, averaging over 10 runs with the shaded areas representing two standard errors. The depicted return is averaged over 500 episodes and the curves are smoothed using an exponential average. Notice that with a small experience replay buffer, DQN(S) performs much worse than all other algorithms, including MeDQN(U), which also uses a small experience replay buffer of the same size as DQN(S). For tasks with a relatively lower input dimension, such as Asterix and Breakout, both MeDQN(R) and MeDQN(U) match up with or even outperform DQN. For Seaquest and Space Invaders which have larger input dimensions, MeDQN(R) is comparable with DQN even though it uses a much smaller experience replay buffer (10% of the replay buffer size in DQN). Meanwhile, the performance of MeDQN(U) is significantly lower than MeDQN(R), mainly due to different state sampling strategies. As discussed in Section 3.3.3, when the state space $\mathcal{S}$ is too large as the cases for Seaquest and Space Invaders, states generated with uniform state sampling can not cover visited states well enough, resulting in poor knowledge consolidation. In this circumstance, storing and sampling real states is usually a better approach. Overall, we conclude that MeDQN(R) is more memory-efficient than DQN while achieving comparably high performance and high sample efficiency.

3.4.5 An Ablation Study of Knowledge Consolidation

In this section, we present an ablation study of knowledge consolidation in Seaquest. By setting $\lambda_{start} = \lambda_{end} = 0$ in MeDQN(R), we removed the consolidation loss from the training loss and MeDQN(R) is reduced to DQN with E updates for each mini-batch of transitions. Except this, all other training settings are the same as MeDQN(R) with consolidation. For example, we also tuned E and $C_{current}$ for MeDQN(R) without consolidation. The averaged final returns with standard errors are presented in Table 3.3 which shows that MeDQN(R) performs better with consolidation than

Table 3.2: The hyper-parameters of different algorithms for MinAtar tasks in Figure 3.6.

(a) Asterix (MinAtar)

Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**	**MeDQN(R)**
learning rate	1e-4	3e-5	1e-3	3e-4
experience replay buffer size	1e5	32	32	1e4
E	/	/	2	2
λ_{end}	/	/	2	2
C_{target}	1000	1000	300	1000
$C_{current}$	1	1	16	8

(b) Breakout (MinAtar)

Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**	**MeDQN(R)**
learning rate	1e-3	3e-5	3e-3	3e-4
experience replay buffer size	1e5	32	32	1e4
E	/	/	1	2
λ_{end}	/	/	2	4
C_{target}	1000	1000	300	1000
$C_{current}$	1	1	8	4

(c) Space Invaders (MinAtar)

Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**	**MeDQN(R)**
learning rate	3e-4	3e-5	1e-3	3e-4
experience replay buffer size	1e5	32	32	1e4
E	/	/	1	2
λ_{end}	/	/	4	2
C_{target}	1000	1000	300	1000
$C_{current}$	1	1	32	4

(d) Seaquest (MinAtar)

Hyper-parameter	**DQN**	**DQN(S)**	**MeDQN(U)**	**MeDQN(R)**
learning rate	1e-4	3e-5	3e-3	3e-4
experience replay buffer size	1e5	32	32	1e4
E	/	/	1	1
λ_{end}	/	/	2	4
C_{target}	1000	1000	300	1000
$C_{current}$	1	1	32	4

Table 3.3: An ablation study of knowledge consolidation for MeDQN(R) with different buffer sizes, tested in Seaquest (MinAtar). MeDQN(R) performs better with consolidation than MeDQN(R) without consolidation under different buffer sizes. All final returns are averaged over 10 runs, shown with one standard error.

Buffer Size	**1e5**	**1e4**	**1e3**
with consolidation	25.82 ± 1.69	27.92 ± 1.53	17.22 ± 0.90
w.o. consolidation	20.10 ± 0.71	19.78 ± 1.19	16.04 ± 0.60

MeDQN(R) without consolidation under different buffer sizes. These results proved that knowledge consolidation is the key to the success of MeDQN(R); MeDQN(R) is not simply benefiting from choosing a higher E or a larger $C_{current}$. It is the use of knowledge consolidation that allows MeDQN(R) to have a smaller experience replay buffer.

3.4.6 A Study of Robustness to Different Buffer Sizes

Moreover, we study the performance of different algorithms with different buffer sizes on Seaquest (MinAtar), which has the largest input state space. In Table 3.4, we present the averaged returns at the end of training for MeDQN(R) and DQN with different buffer sizes. Clearly, MeDQN(R) is more robust than DQN to buffers at various scales. A similar study in a low-dimensional task (MountainCar-v0) is shown in Figure 3.7. Note that for MeDQN(U), m is fixed as the mini-batch size 32. DQN performs worse and worse as we decrease the buffer size, while MeDQN(U) achieves high performance with a tiny buffer.

Table 3.4: A study of robustness to different buffer sizes, tested in Seaquest (MinAtar). MeDQN(R) is more robust than DQN to buffers at various scales. All final returns are averaged over 10 runs, shown with one standard error.

buffer size	**1e5**	**1e4**	**1e3**
MeDQN(R)	25.82 ± 1.69	27.92 ± 1.53	17.22 ± 0.90
DQN	20.48 ± 0.75	21.38 ± 0.80	15.04 ± 0.66

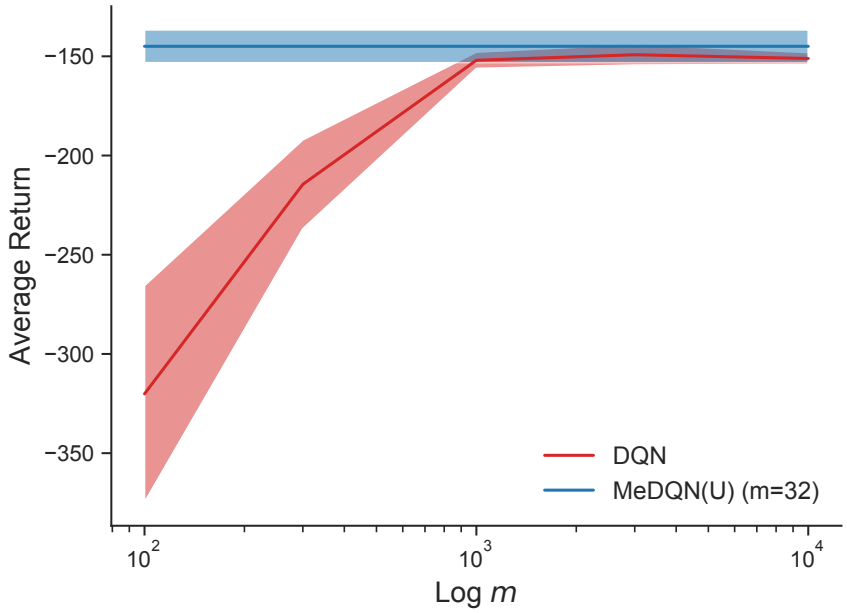


Figure 3.7: A study of robustness to different buffer sizes m in MountainCar-v0. For MeDQN(U), m is fixed as the mini-batch size 32. DQN performs worse and worse as we decrease the buffer size while MeDQN(U) achieves high performance with a tiny buffer. All results are averaged over 20 runs, with the shaded area representing two standard errors.

3.4.7 Additional Results in Atari Games

To further evaluate our algorithm, we conduct experiments in Atari games (Bellemare et al. 2013). Specifically, we select five representative games recommended by Aitchison et al. (2023), including Battlezone, Double Dunk, Name This Game, Phoenix, and Qbert. The input dimensions for all five games are 84×84.

We compare our algorithm MeDQN(R) and DQN with different buffer sizes. Specifically, we use the implementation of DQN in Tianshou (Weng et al. 2022) and implemented MeDQN(R) based on Tianshou's DQN.[2] For DQN, we use the default hyper-parameters of Tianshou's DQN. For MeDQN, $\lambda_{start} = 0.01$; λ_{end} is chosen from $\{2, 4\}$; $E = 1$. We choose $C_{current}$ in $\{20, 40\}$ while the default $C_{current} = 10$ in Tianshou's DQN. Except those hyper-parameters, other hyper-parameters of MeDQN are the same as the default hyper-parameters of Tianshou's DQN.

[2] https://github.com/thu-ml/tianshou/blob/v0.4.10/examples/atari/atari_dqn.py

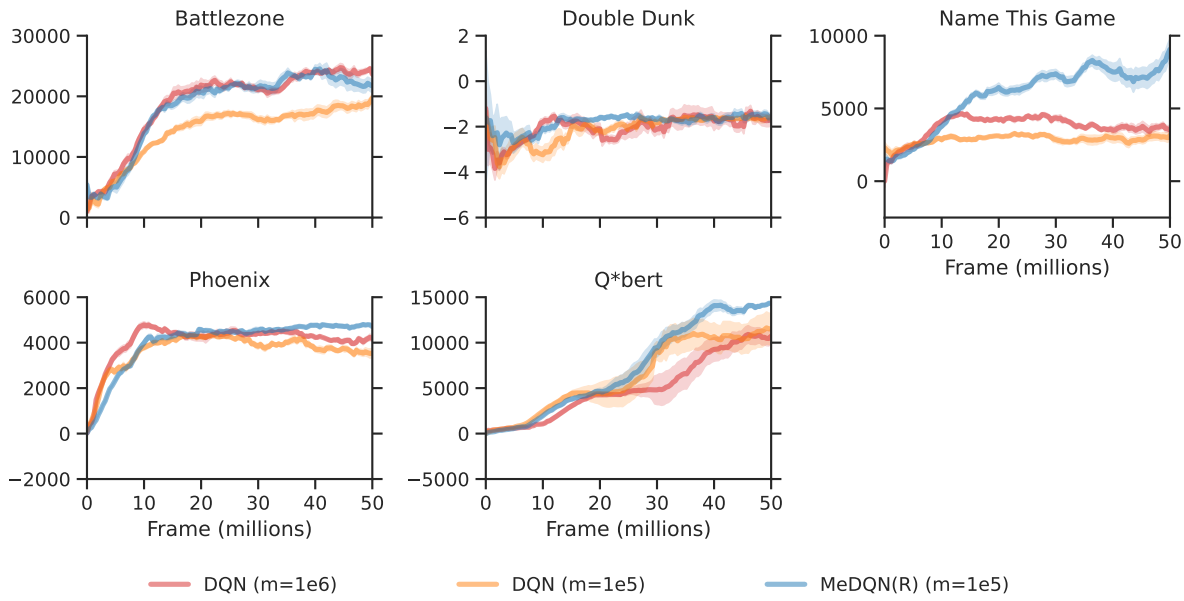


Figure 3.8: The return curves of various algorithms in five Atari tasks over 50 million training frames, with different buffer sizes m. Solid lines correspond to the median performance over 5 random seeds, and the shaded areas correspond to standard errors. Though using a much smaller buffer, MeDQN(R) ($m = 1e5$) outperforms DQN ($m = 1e6$) in three tasks significantly (i.e., Name This Game, Phoenix, and Qbert) and matches DQN's performance in the other two tasks.

We train all algorithms for 50M frames (i.e., 12.5M steps) and summarize results across over 5 random seeds in Figure 3.8. Note that we perform grid search to find the best hyper-parameters. Then we rerun with the best hyper-parameters, using different random seeds. The solid lines correspond to the median performance over 5 random seeds, while the shaded areas represent standard errors. Though using a much smaller buffer, MeDQN(R) ($m = 1e5$) outperforms DQN ($m = 1e6$) in three tasks significantly (i.e., Name This Game, Phoenix, and Qbert) and matches DQN's performance in the other two tasks. Overall, the memory usage of a replay buffer is reduced from 7GB to 0.7GB without hurting the agent's performance, confirming our previous claim that MeDQN(R) is more memory-efficient than DQN while achieving comparably high performance.

3.5 Conclusion

In this chapter, we proposed two memory-efficient algorithms based on DQN. Our algorithms can find a good trade-off between learning new knowledge and preserving old knowledge. By conducting rigorous experiments, we showed that a large experience replay buffer could be successfully replaced by (real or uniform) state sampling, which costs less memory while achieving good performance and high sample efficiency. Considering all above results, we suggest using MeDQN(U) for low-dimensional tasks and MeDQN(R) for high-dimensional tasks.

Chapter 4

Efficient Reinforcement Learning by Reducing Forgetting with Elephant Activation Functions

The previous chapter shows a data-driven approach to reduce forgetting. In contrast, in this chapter, we present a method to mitigate forgetting from an architectural approach. Specifically, researchers have made significant progress in mitigating catastrophic forgetting and proposed many effective methods in recent years, such as optimization-based methods (Farajtabar et al. 2020), regularization-based methods (Riemer et al. 2018), parameter-isolation methods (Mendez et al. 2020), and replay methods (Mendez et al. 2022). Most of these methods tackle the forgetting problem from the algorithmic approach while the other approach, alleviating forgetting by designing neural networks with specific properties, is less explored. There is still a limited understanding of what properties of neural networks lead to catastrophic forgetting. Recently, Mirzadeh et al. (2022a) find that the width of a neural network significantly affects forgetting and provide explanations from the perspectives of gradient orthogonality, gradient sparsity, and lazy training regime. Furthermore, Mirzadeh et al. (2022b) study the forgetting issue on large-scale benchmarks with various network architectures, demonstrating that architectures can also play an important role in reducing forgetting.

The interaction between catastrophic forgetting and neural network architectures remains underexplored. In this chapter, we aim to better understand this interaction by studying the impact of various architectural choices of neural networks on catastrophic forgetting in single-task RL, where many of the continual learning issues appear. Specifically, we focus on activation functions, one of the most important elements in neural networks. Theoretically, we investigate the role of activation functions in the training dynamics of neural networks. Experimentally, we study the effect of various activation functions on catastrophic forgetting under the setting of continual supervised learning. These results suggest that not only sparse representations but also sparse gradients are essential for mitigating forgetting. Based on this discovery, we develop a new type of activation function called *elephant activation functions*. They can generate sparse function values and gradients, thus enhancing neural networks' resilience to catastrophic forgetting. To verify the effectiveness of our method, we substitute classical activation functions with elephant activation functions in neural networks for RL agents and test them across a range of tasks, from basic Gymnasium tasks (Towers et al. 2023) to intricate Atari games (Mnih et al. 2013). The experimental results demonstrate that, under extreme memory constraints, integrating elephant activation functions into RL agents alleviates the forgetting issue, leading to comparable or improved performance while significantly enhancing memory efficiency. Moreover, we show that even when large replay buffers are utilized, applying elephant activation functions remains beneficial, substantially boosting the learning performance.

4.1 Understanding the Success and Failure of Sparse Representation

Firstly, we aim to understand the effectiveness of sparse representations in catastrophic forgetting. To be specific, we argue that sparse representations are effective in reducing forgetting in linear function approximations but are less useful in nonlinear function approximations.

Deep neural networks can automatically generate effective representations (a.k.a. features) to extract key properties from input data. In particular, we call a set of representations sparse when only a small part of representations is non-zero for a given input. It is known that sparse repre-

sentations reduce forgetting and interference in both continual supervised learning and RL (Shen et al. 2021, Liu et al. 2019b). Formally, let $\boldsymbol{x}$ be an input and ϕ be an encoder that transforms the input $\boldsymbol{x}$ into its representation $\phi(\boldsymbol{x})$. The representation $\phi(\boldsymbol{x})$ is sparse when most of its elements are zeros.

First, consider the case of linear approximations. A linear function is defined as $f_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{w}^\top \phi(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}$, where $\boldsymbol{x} \in \mathbb{R}^n$ is an input, $\phi : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a fixed encoder, and $\boldsymbol{w} \in \mathbb{R}^m$ is a weight vector. Assume the representation $\phi(\boldsymbol{x})$ is sparse and non-zero (i.e., $\|\phi(\boldsymbol{x})\|_2 > 0$) for $\boldsymbol{x} \in \mathbb{R}^n$. Next, we show that both Property 2.1 and Property 2.3 are satisfied in this case. Easy to see $\nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}) = \phi(\boldsymbol{x})$. Together with Equation (2.15), we have

$$f_{\boldsymbol{w}'}(\boldsymbol{x}) - f_{\boldsymbol{w}}(\boldsymbol{x}) = -\alpha \nabla_f L(f, F, \boldsymbol{x}_t)\, \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}_t). \tag{4.1}$$

By assumption, $f_{\boldsymbol{w}}(\boldsymbol{x}_t) \neq F(\boldsymbol{x}_t)$ and $\nabla_f L(f, F, \boldsymbol{x}_t) \neq 0$. Then Property 2.1 holds since $f_{\boldsymbol{w}'}(\boldsymbol{x}_t) - f_{\boldsymbol{w}}(\boldsymbol{x}_t) = -\alpha \nabla_f L(f, F, \boldsymbol{x}_t)\, \|\phi(\boldsymbol{x}_t)\|_2^2 \neq 0$. Moreover, it is very likely that $\langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_t) \rangle \approx 0$ when $\boldsymbol{x} \neq \boldsymbol{x}_t$, due to the sparsity of $\phi(\boldsymbol{x})$ and $\phi(\boldsymbol{x}_t)$. Thus, $f_{\boldsymbol{w}'}(\boldsymbol{x}) - f_{\boldsymbol{w}}(\boldsymbol{x}) = -\alpha \nabla_f L(f, F, \boldsymbol{x}_t)\, \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}_t) \rangle \approx 0$ and Property 2.3 holds. We conclude that sparse representations successfully mitigate catastrophic forgetting in linear approximations.

However, for nonlinear approximations, sparse representations can no longer guarantee Property 2.3. Consider a multilayer perceptron (MLP) with one hidden layer $f_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{u}^\top \sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b}) : \mathbb{R}^n \mapsto \mathbb{R}$, where σ is a non-linear activation function, $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{u} \in \mathbb{R}^m$, $\mathbf{V} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, and $\boldsymbol{w} = \{\boldsymbol{u}, \mathbf{V}, \boldsymbol{b}\}$. We compute the NTK in this case, resulting in the following lemma.

Lemma 4.1 (NTK in non-linear approximations)**.** *Given a non-linear function $f_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{u}^\top \sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b}) : \mathbb{R}^n \mapsto \mathbb{R}$, where σ is a non-linear activation function, $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{u} \in \mathbb{R}^m$, $\mathbf{V} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, and $\boldsymbol{w} = \{\boldsymbol{u}, \mathbf{V}, \boldsymbol{b}\}$. The NTK of this nonlinear function is*

$$\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle = \sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})^\top \sigma(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}) + \boldsymbol{u}^\top \boldsymbol{u} (\boldsymbol{x}^\top \boldsymbol{x}_t + 1) \sigma'(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})^\top \sigma'(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}),$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product or Frobenius inner product depending on the context.

Proof. Let $\circ$ denote Hadamard product. By definition, we have

$$
\begin{aligned}
&\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle \\
&= \langle \nabla_u f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_u f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle + \langle \nabla_V f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_V f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle + \langle \nabla_b f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_b f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle \\
&= \langle \sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b}), \sigma(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}) \rangle + \langle \boldsymbol{u} \circ \sigma'(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})\boldsymbol{x}^\top, \boldsymbol{u} \circ \sigma'(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b})\boldsymbol{x}_t^\top \rangle \\
&+ \langle \boldsymbol{u} \circ \sigma'(\mathbf{V}\boldsymbol{x} + \boldsymbol{b}), \boldsymbol{u} \circ \sigma'(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}) \rangle \\
&= \sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})^\top \sigma(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}) + (\boldsymbol{x}^\top \boldsymbol{x}_t + 1) \left(\boldsymbol{u} \circ \sigma'(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})\right)^\top \left(\boldsymbol{u} \circ \sigma'(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b})\right), \\
&= \sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})^\top \sigma(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}) + \boldsymbol{u}^\top \boldsymbol{u}(\boldsymbol{x}^\top \boldsymbol{x}_t + 1)\sigma'(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})^\top \sigma'(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}).
\end{aligned}
$$

□

Note that the encoder ϕ is no longer fixed, and we have $\phi_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})$, where $\boldsymbol{\theta} = \{\mathbf{V}, \boldsymbol{b}\}$ are learnable parameters. By Lemma 4.1,

$$
\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle = \phi_{\boldsymbol{\theta}}(\boldsymbol{x})^\top \phi_{\boldsymbol{\theta}}(\boldsymbol{x}_t) + \boldsymbol{u}^\top \boldsymbol{u}(\boldsymbol{x}^\top \boldsymbol{x}_t + 1)\phi'_{\boldsymbol{\theta}}(\boldsymbol{x})^\top \phi'_{\boldsymbol{\theta}}(\boldsymbol{x}_t). \tag{4.2}
$$

Compared with the NTK in linear approximations (Equation (4.1)), Equation (4.2) has an additional term $\boldsymbol{u}^\top \boldsymbol{u}(\boldsymbol{x}^\top \boldsymbol{x}_t + 1)\phi'_{\boldsymbol{\theta}}(\boldsymbol{x})^\top \phi'_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$, due to a learnable encoder $\phi_{\boldsymbol{\theta}}$. With sparse representations, we have $\phi_{\boldsymbol{\theta}}(\boldsymbol{x})^\top \phi_{\boldsymbol{\theta}}(\boldsymbol{x}_t) \approx 0$. However, it is not necessarily true that $\boldsymbol{u}^\top \boldsymbol{u}(\boldsymbol{x}^\top \boldsymbol{x}_t + 1)\phi'_{\boldsymbol{\theta}}(\boldsymbol{x})^\top \phi'_{\boldsymbol{\theta}}(\boldsymbol{x}_t) \approx 0$ even when $\boldsymbol{x}$ and $\boldsymbol{x}_t$ are quite dissimilar, which violates Property 2.3. For example, when Tanh is used as the activation function, for $x_1 = 0, x_2 > 0$, we have $\mathrm{Tanh}(x_1)\,\mathrm{Tanh}(x_2) = 0$ while $\mathrm{Tanh}'(x_1)\,\mathrm{Tanh}'(x_2) > 0$. To conclude, our analysis indicates that sparse representations alone are not enough to reduce forgetting in nonlinear approximations.

4.2 Obtaining Sparsity with Elephant Activation Functions

Although Lemma 4.1 shows that the forgetting issue can not be fully addressed with sparse representations solely in deep learning methods, it also points out a possible solution: sparse gradients. With sparse gradients, we could have $\phi'_{\boldsymbol{\theta}}(\boldsymbol{x})^\top \phi'_{\boldsymbol{\theta}}(\boldsymbol{x}_t) \approx 0$. Together with sparse representations, we

Table 4.1: The function sparsity and gradient sparsity of various activation functions. Among them, only Elephant is sparse in terms of both function values and gradient values, according to Definition 4.1 and Lemma 4.2.

Activation	Function Sparsity	Gradient Sparsity
ReLU	1/2	1/2
Sigmoid	1/2	1
Tanh	0	1
ELU	0	1/2
Elephant	1	1

may still satisfy Property 2.3 in nonlinear approximations and thus reduce more forgetting. Specifically, we aim to design new activation functions to obtain both sparse representations and sparse gradients.

To begin with, we first define the sparsity of a function, which also applies to activation functions.

Definition 4.1 (Sparse Function)**.** *For a function $\sigma : \mathbb{R} \mapsto \mathbb{R}$, we define the sparsity of function σ on input domain $[-C, C]$ as*

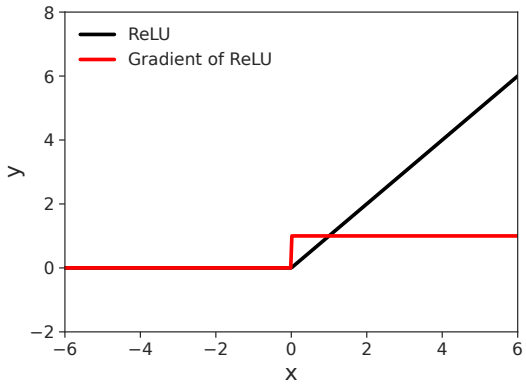
$$S_{\epsilon,C}(\sigma) = \frac{|\{x \mid |\sigma(x)| \leq \epsilon, x \in [-C, C]\}|}{|\{x \mid x \in [-C, C]\}|} = \frac{|\{x \mid |\sigma(x)| \leq \epsilon, x \in [-C, C]\}|}{2C},$$

where ϵ is a small positive number and $C > 0$. As a special case, when $\epsilon \to 0^+$ and $C \to \infty$, define

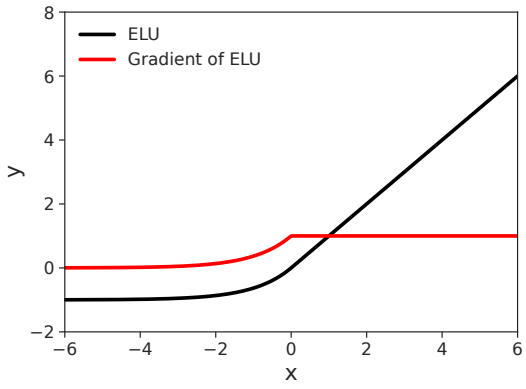
$$S(\sigma) = \lim_{\epsilon \to 0^+} \lim_{C \to \infty} S_{\epsilon,C}(\sigma).$$

We call σ a $S(\sigma)$-sparse function. In particular, σ is called a sparse function when $S(\sigma) = 1$.

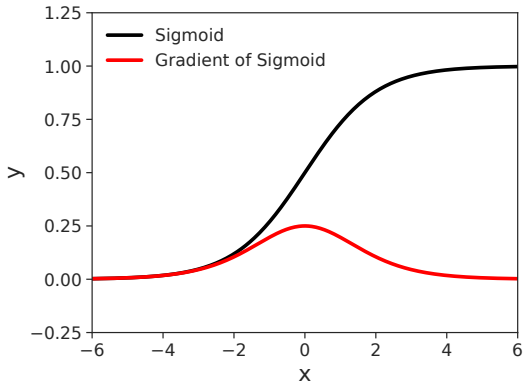
Easy to verify that $0 \leq S(\sigma) \leq 1$. The sparsity of a function shows the fraction of nearly zero outputs given a symmetric input domain. For example, neither $\mathrm{ReLU}(x)$ nor $\mathrm{ReLU}'(x)$ is a sparse functions. $\mathrm{Tanh}(x)$ is not a sparse function while $\mathrm{Tanh}'(x)$ is a sparse function. In Figure 4.1 and Table 4.1, we present more examples as well as visualizations of the activation functions and their gradients.



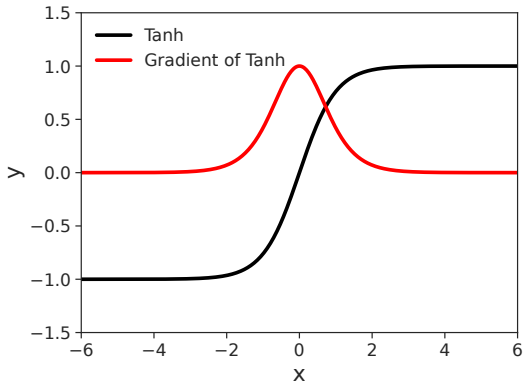
(a) ReLU



(b) ELU

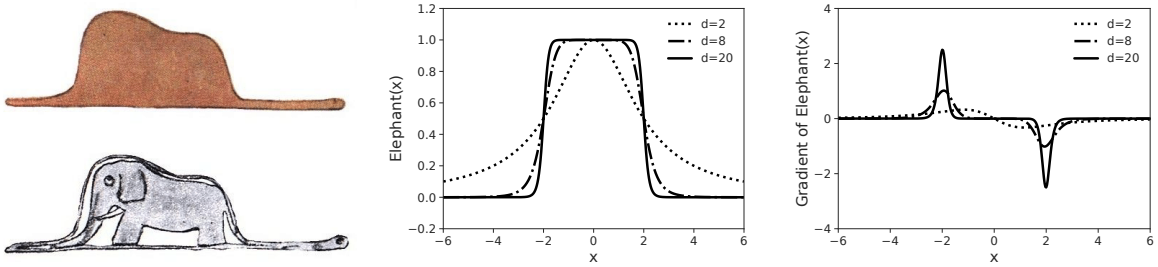


(c) Sigmoid



(d) Tanh

Figure 4.1: Visualizations of common activation functions and their gradients.



(a) A drawing of an elephant. (b) Elephant(x) (c) Elephant$'(x)$

Figure 4.2: (a) "My drawing was not a picture of a hat. It was a picture of a boa constrictor digesting an elephant." *The Little Prince*, by Antoine de Saint Exupéry. (b) Elephant functions for $a = 2$, $h = 1$, and various d. (c) Gradients of elephant functions for $a = 2$, $h = 1$, and various d.

It is worth noting that sparse activation functions and sparse representations are not the same—sparse activation functions are one-to-one mappings while sparse representations are vectors in which most elements are zeros. By incorporating sparse activation functions in neural networks, we increase the likelihood of generating sparse representations given diverse inputs.

Next, we propose a novel class of bell-shaped activation functions, *elephant activation functions*.[1] Formally, an elephant function is defined as

$$\text{Elephant}(x) = \frac{h}{1 + \left|\frac{x}{a}\right|^d}, \tag{4.3}$$

where a controls the width of the function, h is the height, and d controls the slope. We call a neural network that uses elephant activation functions an *elephant neural network (ENN)*. For example, for MLPs, we have *elephant MLPs (EMLPs)* correspondingly.

As shown in Figure 4.2, elephant activation functions have both sparse function values and sparse gradient values, which can be formally proved as well. Note that the radial basis functions have a similar shape to elephant functions. However, elephant functions are sparser than radial basis functions, as the gradient of an elephant function near 0 is flatter than that of a radial basis function.

[1]We name this bell-shaped activation function the *elephant* function, as it suggests that this activation empowers neural networks with continual learning ability, echoing the saying "an elephant never forgets." The bell shape also resembles the silhouette of an elephant (see Figure 4.2), paying homage to *The Little Prince* by Antoine de Saint Exupéry.

Lemma 4.2. Elephant(x) *and* Elephant$'(x)$ *are sparse functions.*

Proof. Without loss of generality, we set $h = 1$. So $\text{Elephant}(x) = \frac{1}{1+|\frac{x}{a}|^d}$ and $\left|\text{Elephant}'(x)\right| = \frac{d}{a}\left|\frac{x}{a}\right|^{d-1}(\frac{1}{1+\left|\frac{x}{a}\right|^d})^2$. For $0 < \epsilon < 1$, easy to verify that

$$|x| \geq a(\frac{1}{\epsilon} - 1)^{1/d} \Longrightarrow \text{Elephant}(x) \leq \epsilon \quad \text{and} \quad |x| \geq \frac{d}{2\epsilon} \Longrightarrow \left|\text{Elephant}'(x)\right| \leq \epsilon.$$

For $C > a(\frac{1}{\epsilon} - 1)^{1/d}$, we have $S_{\epsilon,C}(\text{Elephant}) \geq \frac{C - a(\frac{1}{\epsilon}-1)^{1/d}}{C}$, thus

$$S(\text{Elephant}) = \lim_{\epsilon\to 0^+}\lim_{C\to\infty} S_{\epsilon,C}(\text{Elephant}) \geq \lim_{\epsilon\to 0^+}\lim_{C\to\infty} \frac{C - a(\frac{1}{\epsilon} - 1)^{1/d}}{C} = \lim_{\epsilon\to 0^+} 1 = 1.$$

Similarly, for $C > \frac{d}{2\epsilon}$, we have $S_{\epsilon,C}(\text{Elephant}') \geq \frac{C - \frac{d}{2\epsilon}}{C}$, thus

$$S(\text{Elephant}') = \lim_{\epsilon\to 0^+}\lim_{C\to\infty} S_{\epsilon,C}(\text{Elephant}') \geq \lim_{\epsilon\to 0^+}\lim_{C\to\infty} \frac{C - \frac{d}{2\epsilon}}{C} = \lim_{\epsilon\to 0^+} 1 = 1.$$

Note that $S(\text{Elephant}) \leq 1$ and $S(\text{Elephant}') \leq 1$. Together, we conclude that $S(\text{Elephant}) = 1$ and $S(\text{Elephant}') = 1$; Elephant(x) and Elephant$'(x)$ are sparse functions. □

Specifically, d controls the sparsity of gradients for elephant functions. The larger the value of d, the sharper the slope and the sparser the gradient. On the other hand, a controls the sparsity of the function itself. Since both the gradient of an elephant function and the elephant function itself are sparse functions, we increase the likelihood of both sparse representations and gradients (see Equation (4.2)). Formally, we show that Property 2.3 holds under certain conditions for elephant functions.

Theorem 4.1. *Define $f_{\boldsymbol{w}}(\boldsymbol{x})$ as in Lemma 4.1. Let σ be the elephant activation function with $h = 1$ and $d \to \infty$. When $|\mathbf{V}(\boldsymbol{x} - \boldsymbol{x}_t)| \succ 2a\mathbf{1}_m$, we have $\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t)\rangle = 0$, where $\succ$ denotes an element-wise inequality symbol and $\mathbf{1}_m = [1, \cdots, 1]^\top \in \mathbb{R}^m$.*

Proof. When $d \to \infty$, the elephant function is a rectangular function, i.e.,

$$\sigma(x) = \mathrm{rect}(x) = \begin{cases} 1, & |x| < a, \\ \frac{1}{2}, & |x| = a, \\ 0, & |x| > a. \end{cases}$$

In this case, it is easy to verify that $\forall\, x, y \in \mathbb{R}$, $|x - y| > 2a$, we have $\sigma(x)\sigma(y) = 0$ and $\sigma'(x)\sigma'(y) = 0$. Denote $\Delta_{\boldsymbol{x}} = \boldsymbol{x} - \boldsymbol{x}_t$. Then when $|\mathbf{V}\Delta_{\boldsymbol{x}}| \succ 2a\mathbf{1}_m$, we have $\sigma(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})^\top \sigma(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}) = 0$ and $\sigma'(\mathbf{V}\boldsymbol{x} + \boldsymbol{b})^\top \sigma'(\mathbf{V}\boldsymbol{x}_t + \boldsymbol{b}) = 0$. In other words, when $\boldsymbol{x}$ and $\boldsymbol{x}_t$ are dissimilar in the sense that $|\mathbf{V}(\boldsymbol{x} - \boldsymbol{x}_t)| \succ 2a\mathbf{1}_m$, we have $\langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(\boldsymbol{x}_t) \rangle = 0$. □

Theorem 4.1 mainly proves that when $d \to \infty$, Property 2.3 holds with elephant functions. However, even when d is a small integer (e.g., 8), we can still obtain this property, as we will show in the experiment section.

4.3 Related Work

4.3.1 Architecture-Based Continual Learning

Mirzadeh et al. (2022a;b) are particularly notable for studying the effect of network architectures on continual learning. Several approaches involve the selection and allocation of a subset of weights in a network for each task (Ammar et al. 2014, Mallya and Lazebnik 2018, Sokar et al. 2021, Fernando et al. 2017, Serra et al. 2018, Masana et al. 2021, Li et al. 2019, Yoon et al. 2018, Mendez et al. 2020), or the allocation of a specific network to each task (Rusu et al. 2016, Aljundi et al. 2017). Some methods expand networks dynamically (Yoon et al. 2018, Hung et al. 2019, Ostapenko et al. 2019) as training continues. Inspired by biological neural circuits, Shen et al. (2021), Bricken et al. (2023), and Madireddy et al. (2023) propose novel networks which generate sparse representations by design. Finally, our method is closely related to sparse activation functions, such as fuzzy tiling activation (FTA) (Pan et al. 2022a), Maxout (Goodfellow et al. 2013), and local winner-take-all

(LWTA) (Srivastava et al. 2013). Specifically, inspired by tile coding (Sutton and Barto 2018), FTA maps a scalar to a vector with a controllable sparsity level. Maxout outputs the maximum value across k input features. LWTA is similar to Maxout with a slight difference: while Maxout only selects and outputs maximum values, LWTA selects maximum values, sets others to zeros, and then outputs them together. Compared to these activation functions, Elephant is simpler to implement and can generate both sparse representations and gradients, as supported by our theoretical and experimental results.

4.3.2 Sparsity in Deep Learning

Sparse representations are known to help reduce forgetting for decades (French 1992). In supervised learning, dynamic sparse training (Dettmers and Zettlemoyer 2019, Liu et al. 2020, Sokar et al. 2021), dropout variants (Srivastava et al. 2013, Goodfellow et al. 2013, Mirzadeh et al. 2020, Abbasi et al. 2022, Sarfraz et al. 2023), and pruning methods (Guo et al. 2016, Frankle and Carbin 2019, Blalock et al. 2020, Zhou et al. 2020, Wang et al. 2022) are shown to speed up training and improve generalization. Additionally, Lee et al. (2021), Sokar et al. (2022), and Tan et al. (2023) propose dynamic sparse training approaches for RL which achieve comparable or improved performance, higher sample efficiency, and better computation efficiency. Furthermore, Le et al. (2017) and Liu et al. (2019b) show that sparse representations stabilize training and improve performance in RL. Finally, Ceron et al. (2024) demonstrates that increasing network sparsity by gradual magnitude pruning improves the learning performance of value-based RL agents. Our approach differs from them in its simplicity and effectiveness—by simply replacing classical activation functions with Elephant, we can enhance the efficiency of training RL agents.

4.3.3 Local Elasticity and Memorization

He and Su (2020) propose the concept of local elasticity. Chen et al. (2020) introduce label-aware neural tangent kernels, showing that models trained with these kernels are more locally elastic. Mehta et al. (2021) prove a theoretical connection between the scale of network initialization and

local elasticity, demonstrating extreme memorization using large initialization scales. Incorporating Fourier features in the input of a network also induces local elasticity, which is greatly affected by the initial variance of the Fourier basis (Li and Pathak 2021).

4.4 Experiments

In this section, we experimentally validate a series of hypotheses regarding the effectiveness of elephant activation functions under regression and RL settings.

For elephant activation functions, we use the following setup unless explicitly stated otherwise. Specifically, since the activation area of Elephant is narrow and centered around zero (i.e., from $-a$ to a), we apply layer normalization (Ba et al. 2016) to normalize the input vector in order to get a better trade-off between stability and plasticity (Mermillod et al. 2013, Lin et al. 2022, Jung et al. 2023, Lyle et al. 2023). We also make some parameters of Elephant adaptive and learnable, assigning an individual elephant activation function to each unit of a neural network. Particularly, all elephant functions are initialized with the same hyper-parameters at the beginning of training. During training, we optimize h and a using gradient descent while keeping d fixed for each elephant function (see Equation (4.3)). Compared to fixing h and a during training, this approach is shown to match or even improve performance in our preliminary experiments. Furthermore, when Elephant is applied, to improve the diversity of initially generated features, we initialize bias values in a linear layer with evenly spaced numbers over the interval $[-\sqrt{3}\sigma_{bias}, \sqrt{3}\sigma_{bias}]$, where σ_{bias} is a positive hyper-parameter. When other activation functions are used, bias values are initialized with zeros following PyTorch's default setting. For weight values in a linear layer, we follow the default initialization as PyTorch by sampling weight values from a uniform distribution $U[-\sqrt{k}, \sqrt{k}]$, where $k = 1/\text{in\_features}$.

4.4.1 Streaming Learning for Regression

RL is relatively complex due to agent-environment interactions. Instead, we first perform experiments in a simple regression task in the streaming learning setting to answer the following question:

Can we achieve mild forgetting and local elasticity with elephant activation functions?

In this setting, a learning agent is presented with one sample only at each time step and then performs learning updates. Moreover, the learning happens in a single pass of the whole dataset; that is, each sample only occurs once. Furthermore, the data stream is assumed to be non-independent and identically distributed (non-IID). Finally, the evaluation happens after each new sample arrives, which requires the agent to learn quickly while avoiding forgetting. Streaming learning methods enable real-time adaptation; thus, they are more suitable in real-world scenarios where data is received in a continuous flow.

We consider approximating a sine function in the streaming learning setting. In this task, there is a stream of data $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$, where $0 \leq x_1 < x_2 < \cdots < x_n \leq 2$, $y_i = \sin(\pi x_i)$, and $n = 200$. The learning agent f is an MLP with one hidden layer of size $1,000$. At each time step t, the agent only receives one sample (x_t, y_t). We minimize the loss $l_t = (f(x_t) - y_t)^2$, where $f(x_t)$ is the agent's prediction. For each new arriving example, we perform 10 updates with Adam (Kingma and Ba 2015) optimizer. The best learning rate is selected from $\{3e-3, 1e-3, 3e-4, 1e-4, 3e-5, 1e-5\}$. For Elephant, we set $d = 8$, $h = 1$, $a = 0.08$, and $\sigma_{bias} = 1.28$. For clarity of demonstration, we omit layer normalization before applying Elephant and fix h and a during training, avoiding their influence on the visualization of the NTK functions. For SR-NN, we apply various classical activation functions (ReLU, Sigmoid, ELU, and Tanh), tune Set KL loss weight λ and β (see Liu et al. (2019b) for details), and present the best result. Specifically, we choose λ from $\{0, 0.1, 0.01, 0.001\}$ and choose β from $\{0.05, 0.1, 0.2\}$. We measure the agent performance by the mean square error (MSE) on a test dataset with $1,000$ samples, where the inputs are evenly spaced over the interval $[0, 2]$.

We compare our method Elephant with two kinds of baselines. One is classical activation

Table 4.2: The test MSEs of various methods in streaming learning for a simple regression task. Lower is better. Elephant achieves the best test performance.

Method	Test Performance (MSE)
ReLU	0.4729 ± 0.0110
Sigmoid	0.4583 ± 0.0008
Tanh	0.4461 ± 0.0013
ELU	0.4521 ± 0.0019
SR-NN	0.4061 ± 0.0036
Elephant	$\mathbf{0.0081 \pm 0.0009}$

functions, including ReLU, Sigmoid, ELU, and Tanh. The other is the sparse representation neural network (SR-NN) (Liu et al. 2019b), which generates sparse representations by regularization. We present the test MSEs of various methods in Table 4.2. Lower is better. All results are averaged over 5 runs, reported with standard errors. For SR-NN, we present the best result among the combinations of SR-NNs and classical activation functions.

Clearly, Elephant has the best performance, achieving a test MSE that is two orders of magnitude lower compared to baselines, which have similar orders to each other. Moreover, SR-NN performs slightly better than classical activation functions, showing the benefits of sparse representations. Yet, compared with Elephant, the test MSE of SR-NN is still large, indicating that it fails to approximate well. To analyze in depth, we plot the true function $\sin(\pi x)$, the learned function $f(x)$, and the NTK function $\text{NTK}(x) = \langle \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(x), \nabla_{\boldsymbol{w}} f_{\boldsymbol{w}}(x_t) \rangle$ at different training stages for Elephant and SR-NN in Figure 4.3. The plots of classical activation functions are not presented since they are similar to the plots of SR-NN. We normalize $\text{NTK}(\boldsymbol{x})$ such that the function value is in $[-1, 1]$. The plots in the first row show that for Elephant with $d = 8$, $\text{NTK}(x)$ quickly decreases to 0 as x moves away from x_t, demonstrating the local elasticity of applying Elephant with a small d. However, SR-NNs (and MLPs with classical activation functions) are not locally elastic; the learned function basically evolves as a linear function (Figure 4.3), a phenomenon that often appears in over-parameterized neural networks (Jacot et al. 2018, Chizat et al. 2019).

By injecting local elasticity to a neural network, we can break the inherent global generalization ability (Ghiassian et al. 2020) of the neural networks, constraining the output changes of the neural

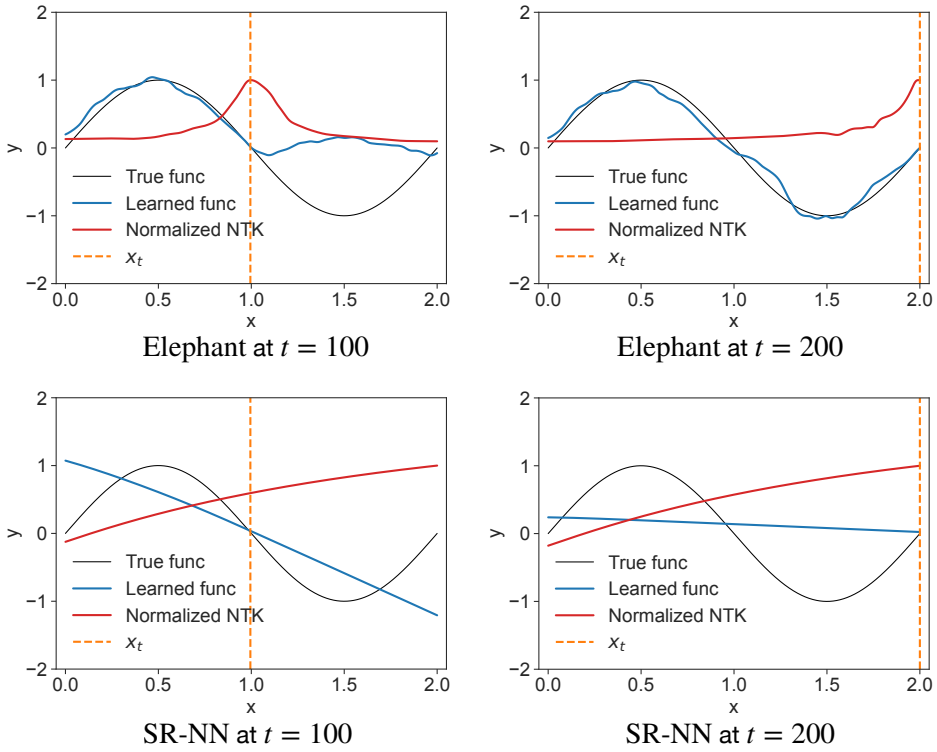
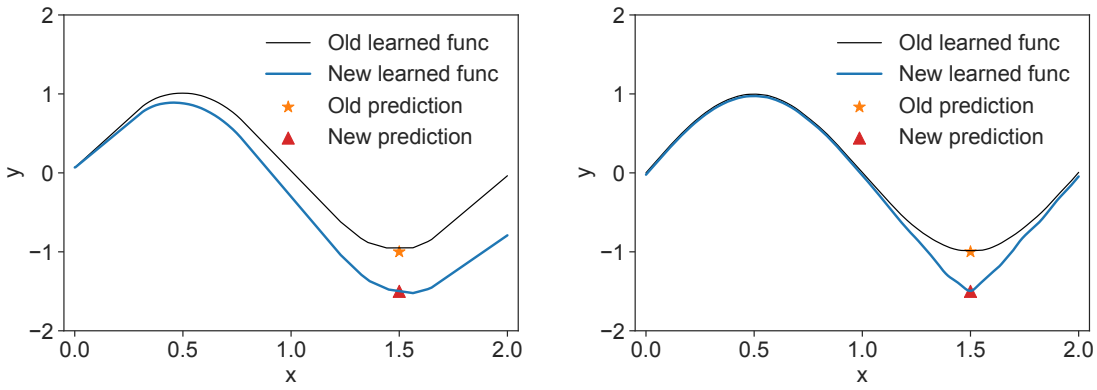


Figure 4.3: Plots of the true function $\sin(\pi x)$, the learned function $f(x)$, and the NTK function $\text{NTK}(x)$ at different training stages using Elephant and SR-NN for approximating a sine function. The $\text{NTK}(x)$ of Elephant quickly reduces to 0 as x moves away from x_t, demonstrating local elasticity.

network to small local areas. Utilizing this phenomenon, we can update a wrong prediction by "editing" outputs of a neural network nearly point-wisely. To verify, we first train a neural network to approximate $\sin(\pi x)$ well enough, calling it the old learned function. Now assume that the original y value of an input x is changed to y', while the true values of other inputs remain the same. Our goal is to update the prediction for input x to y', while keeping the predictions of other inputs without expensive re-training on the whole dataset. Note that this requirement is common in RL (see next section). Specifically, we choose $x = 1.5$, $y = -1.0$, and $y' = -1.5$; and perform experiments with Elephant and ReLU, showing in Figure 4.4. Both methods successfully update the prediction at $x = 1.5$ to y'. However, besides the prediction at $x = 1.5$, the learned function with ReLU is changed globally while the changes with Elephant are mainly confined in a small local area around $x = 1.5$. That is, we can successfully correct the wrong prediction nearly point-wise



(a) ReLU

(b) Elephant

Figure 4.4: Plots of updating a wrong prediction with ReLU and Elephant. Elephant allows updating the old prediction nearly point-wisely by "editing" the output value of the neural network, a capability lacking in classical activation functions (e.g., ReLU).

by "editing" the output value for elephant neural networks (ENNs), but not for classical neural networks.

To conclude, we showed that (1) sparse representations do not suffice to address the forgetting issue, (2) ENNs are locally elastic even when d is small, and (3) ENNs can continually learn to solve regression tasks by reducing forgetting.

4.4.2 Reinforcement Learning

In Figure 3.2, we show that the forgetting issue exists even in single RL tasks and a large replay buffer largely masks it. Without a replay buffer, a single RL task can be viewed as a series of tasks without clear boundaries (Dabney et al. 2021). For example, in temporal difference (TD) learning, the true value function is approximated by V with bootstrapping: $V(s_t) \leftarrow r_{t+1} + \gamma V(s_{t+1})$, where s_t and s_{t+1} are two successive states and $r_{t+1} + \gamma V(s_{t+1})$ is named the TD target. During training, the TD target constantly changes due to bootstrapping, non-stationary state distribution, and changing policy. To speed up learning while reducing forgetting, it is crucial to update $V(S_t)$ to the new TD target without changing other state values too much, where local elasticity can help.

In the following, we aim to demonstrate that incorporating elephant activation functions helps

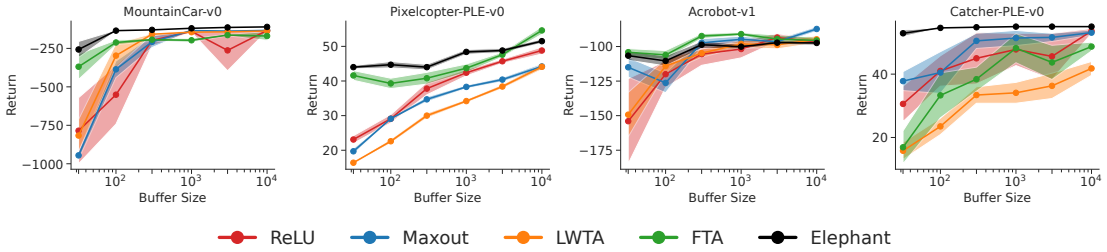


Figure 4.5: The performance of DQN in 4 Gymnasium and PyGame tasks with different activation functions under various buffer sizes, measured as the average return of the last 10% episodes. Elephant demonstrates its robustness to varying buffer sizes.

reduce forgetting in RL, thus improving sample efficiency and learning performance. We mainly consider two representative value-based RL algorithms — deep Q-network (DQN) (Mnih et al. 2013; 2015) and Rainbow (Hessel et al. 2018). We consider other activation functions as baselines, including ReLU, Tanh, Maxout (Goodfellow et al. 2013), LWTA (Srivastava et al. 2013), and FTA (Pan et al. 2022a). While ReLU and Tanh are widely used classical activation functions, Maxout, LWTA, and FTA are designed to generate sparse representations (see Related Work for detailed introductions).

Elephant Improves Memory Efficiency

First, we focus on addressing the following question:

> *Can elephant activation functions help achieve strong performance with a small memory budget?*

Specifically, we implement DQN with JAX (Bradbury et al. 2018) from scratch based on Lan (2019) and test DQN in 4 classical RL tasks (i.e., MountainCar-v0, Acrobot-v1, Catcher, and Pixelcopter) from Gymnasium (Towers et al. 2023) and PyGame Learning Environment (Tasfi 2016) under various buffer sizes. Note that we select these small-scale RL tasks so that we could perform thorough hyperparameter tuning and ensure a fair comparison within a reasonable time frame. For instance, we consider buffer sizes in $\{32, 1e2, 3e2, 1e3, 3e3, 1e4\}$, where the default buffer size is $1e4$. And for each hyper-parameter setup, a best learning rate is selected from $\{1e-2, 3e-$

$3, 1e-3, 3e-4, 1e-4, 3e-5, 1e-5, 3e-6\}$, while RMSProp (Tieleman and Hinton 2012) is applied with a decay rate of 0.999 for optimization. The discount factor $\gamma = 0.99$. The mini-batch size is 32. The default network is an MLP with one hidden layer of size $1,000$ in all tasks. However, when different activation functions are applied, we adjust the width of the hidden layer so that the total number of parameters is close to each other. For Maxout and LWTA, we set $k = 5$. For FTA, we set $k = 20$ and $[l, u] = [-20, 20]$ following Pan et al. (2022a). For Elephant, to make a fair comparison, we use the same set of hyper-parameters for all DQN experiments in classical RL tasks, i.e., $d = 4$, $h = 1$, $a = 0.2$, and $\sigma_{bias} = 0.4$. Note that tuning the hyper-parameters of Elephant for each task and buffer size could further improve the performance.

We plot agents' performance of different activation functions and buffer sizes in Figure 4.5. All results are averaged over 10 runs, and the shaded areas represent standard errors. Clearly, Elephant demonstrates its robustness to varying buffer sizes. When the buffer size is reduced, the performance of Elephant remains high in all four tasks, while the performance of all other activations drops significantly. In summary, these results confirm the effectiveness of Elephant in reducing forgetting and improving memory efficiency for DQN.

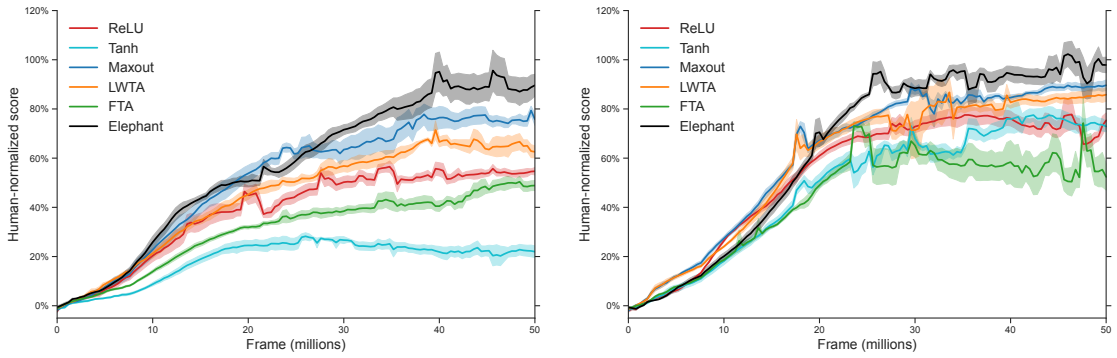
Elephant Improves Sample Efficiency

Next, we investigate the following question:

> *Given a similar amount of training samples and sufficient memory resources, can elephant activation functions outperform classical activation functions?*

To answer this question, we test DQN and Rainbow in 10 Atari tasks (Bellemare et al. 2013) with different activation functions. Specifically, we selected 10 representative Atari tasks recommended by Aitchison et al. (2023)—Amidar, Battlezone, Bowling, Double Dunk, Frostbite, Kung-Fu Master, Name This Game, Phoenix, Q*bert, and River Raid——which produce scores that closely correlate with the performance on the full set of 57 Atari tasks, while requiring substantially less training cost. The implementation of DQN and Rainbow are adapted from Tianshou (Weng et al. 2022). [2]

[2] https://github.com/thu-ml/tianshou/blob/v0.4.10/examples/atari/



(a) DQN (b) Rainbow

Figure 4.6: The learning curves of DQN and Rainbow aggregated over 10 Atari tasks for 6 activation functions. Solid lines correspond to the average performance over 5 runs, while shaded areas correspond to standard errors. Elephant surpasses the baselines significantly.

We also follow the default hyper-parameters and training setups as in Tianshou unless explicitly stated otherwise. The mini-batch size is 32. The discount factor is 0.99. Adam (Kingma and Ba 2015) is applied to optimize. We train all agents for 50M frames (i.e., 12.5M steps). For DQN, the learning rate is $1e-4$ while it is $6.25e-5$ for Rainbow. The buffer size is 1 million.

For Maxout and LWTA, we set $k = 4$. For FTA, we set $k = 20$ and $[l, u] = [-20, 20]$ following Pan et al. (2022a). We adjust the width of the hidden layer so that the total number of parameters is close to each other when various activation functions are applied. To make a fair comparison, we use the same hyper-parameters for Elephant (i.e., $d = 4$, $h = 1$, $a = 0.1$, and $\sigma_{bias} = 2$) in all Atari experiments, although tuning them for each task could further boost the performance. Furthermore, since Rainbow uses noisy linear layers (Fortunato et al. 2018), we follow the default initialization in Rainbow and do not reinitialize the weights and biases when Elephant is applied in Rainbow. Note that for both DQN and Rainbow, we only apply these activation functions to the penultimate layer; the CNN feature network still uses ReLU as the default activation function.

In Figure 4.6, we show the test human-normalized scores of DQN and Rainbow with different activation functions, aggregated over 10 Atari tasks. In particular, the raw score of each task is first normalized so that 0% corresponds to a random agent and 100% to a human expert. Then those human-normalized scores are smoothed with a moving average over 20 subsequent points within

the same run. Smoothed curves are then grouped by frame and seed, and for each frame we take the median across tasks, thereby collapsing task-level variability into a single performance score for each pair of frame and seed. Finally, in Figure 4.6, we plot the means (solid lines) across seeds for frame, with shaded regions denoting the standard errors (shaded areas) of the medians. Moreover, we present the detailed test performance of DQN and Rainbow in all 10 Atari tasks in Table 4.3 and the return curves in Figure 4.7. Together, these results demonstrate that even when a large buffer is used, Elephant still surpasses the baselines significantly.

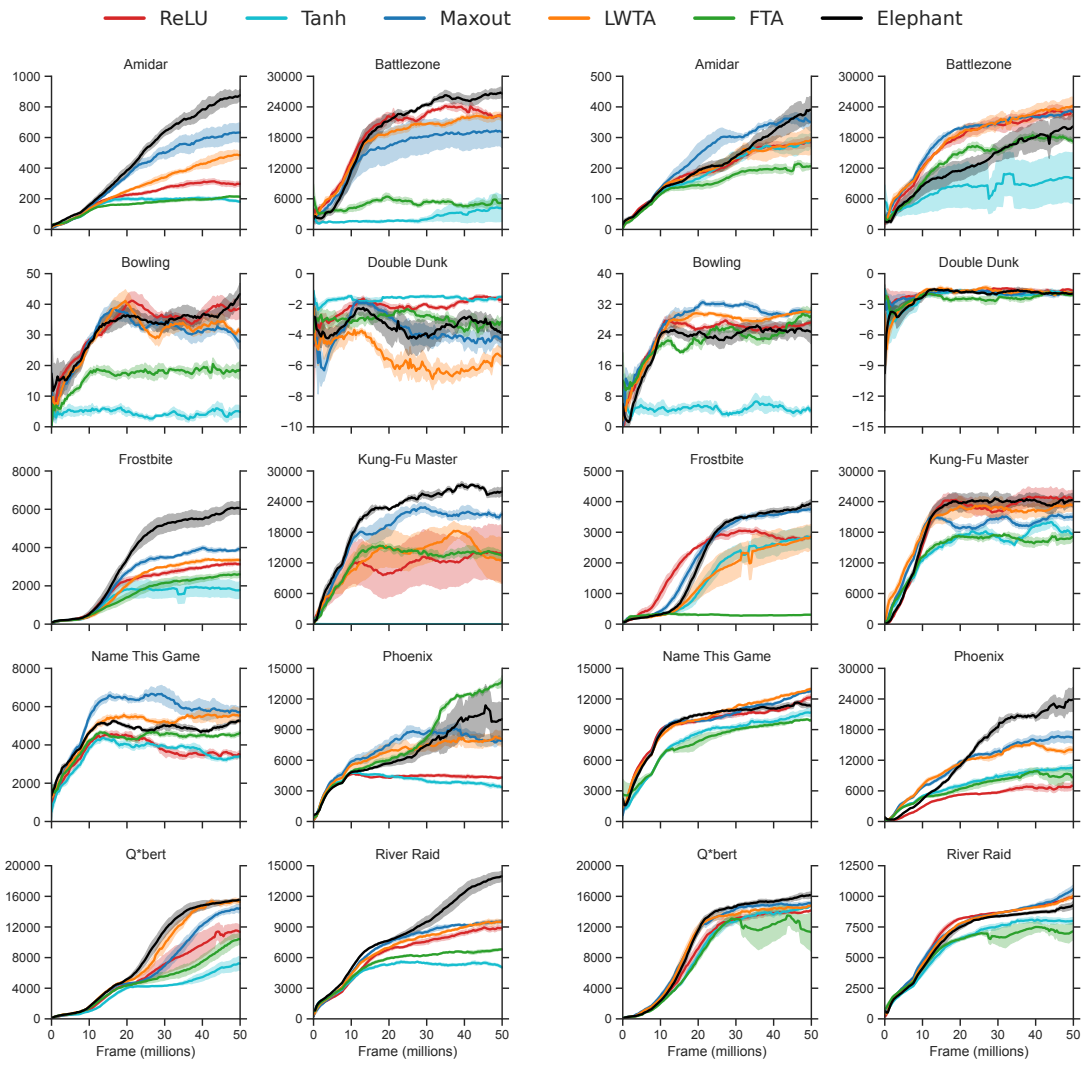
Table 4.3: The performance comparison of DQN and Rainbow with different activation functions across 10 Atari tasks. We report the final test returns averaged over 5 runs as well as the corresponding 95% confidence intervals.

(a) DQN

Task \ **Activation**	ReLU	Tanh	Maxout	LWTA	FTA	Elephant
Amidar	309 ± 30	171 ± 9	620 ± 133	462 ± 40	203 ± 2	912 ± 88
Battlezone	21664 ± 3696	4148 ± 3084	19028 ± 6012	21852 ± 1924	4760 ± 1504	26176 ± 2192
Bowling	44 ± 8	2 ± 2	22 ± 6	33 ± 7	20 ± 7	44 ± 11
Double Dunk	-1.9 ± 0.5	-1.4 ± 0.2	-5.2 ± 2.4	-6.0 ± 1.3	-2.2 ± 0.5	-5.0 ± 1.4
Frostbite	3074 ± 167	1501 ± 628	3734 ± 505	3510 ± 208	2613 ± 433	6001 ± 724
Kung-Fu Master	12996 ± 9809	0 ± 0	22563 ± 2230	10731 ± 7698	12251 ± 3113	28411 ± 2065
Name This Game	3573 ± 484	3889 ± 488	5261 ± 761	5890 ± 551	4751 ± 383	4847 ± 435
Phoenix	4346 ± 246	3060 ± 632	8421 ± 829	7659 ± 1544	14055 ± 1792	11110 ± 2387
Q*bert	11081 ± 1271	7697 ± 1824	14469 ± 898	15228 ± 638	10195 ± 1007	15454 ± 84
River Raid	9560 ± 290	4802 ± 319	9405 ± 243	9669 ± 288	6875 ± 110	14339 ± 1148

(b) Rainbow

Task \ **Activation**	ReLU	Tanh	Maxout	LWTA	FTA	Elephant
Amidar	300 ± 37	285 ± 27	354 ± 64	309 ± 96	211 ± 12	402 ± 71
Battlezone	24104 ± 2224	9844 ± 8764	22320 ± 1712	24308 ± 3284	16176 ± 1520	19600 ± 4088
Bowling	27 ± 3	8 ± 5	31 ± 1	30 ± 1	26 ± 4	24 ± 7
Double Dunk	-1.9 ± 0.6	-1.9 ± 0.4	-2.0 ± 0.3	-1.9 ± 0.6	-2.0 ± 0.3	-2.0 ± 0.3
Frostbite	2825 ± 308	2604 ± 662	3747 ± 399	2706 ± 913	292 ± 17	3961 ± 362
Kung-Fu Master	24583 ± 1642	17225 ± 1413	21275 ± 1324	23064 ± 2794	18502 ± 1282	25421 ± 2004
Name This Game	12321 ± 713	10833 ± 1155	12746 ± 199	13425 ± 734	9616 ± 536	11384 ± 444
Phoenix	6442 ± 474	10649 ± 1170	15024 ± 1894	14408 ± 951	8207 ± 3642	26033 ± 5962
Q*bert	14477 ± 467	14616 ± 514	15537 ± 377	15110 ± 880	11309 ± 4481	16160 ± 794
River Raid	10054 ± 564	8109 ± 455	11127 ± 540	10504 ± 309	7427 ± 979	9563 ± 487



(a) DQN (b) Rainbow

Figure 4.7: The return curves of DQN and Rainbow with various activations in 10 Atari tasks over 50 million training frames during test. Solid lines correspond to the average performance over 5 random seeds, and the shaded areas correspond to standard errors.

Gradient Analysis

In Section 4.2, we claimed that Elephant induces sparse gradient and thus reduces forgetting. Here, we perform a gradient analysis to verify the claim. To be specific, we visualize the gradient covariance matrices at different stages of training DQN in Atari tasks. Essentially, the gradient covariance matrix is a matrix of normalized NTK. Formally, we estimate this matrix (denoted as C) by randomly sampling k training samples $\mathbf{x}_1, \cdots, \mathbf{x}_k$ and compute each element as

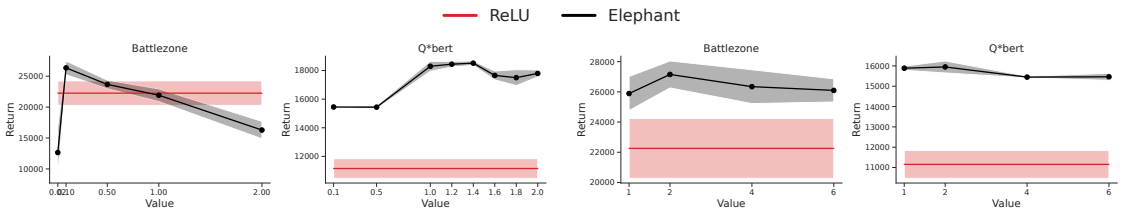
$$C_{ij} = \frac{\langle \nabla_\theta l(\theta, \mathbf{x}_i), \nabla_\theta l(\theta, \mathbf{x}_j) \rangle}{\|\nabla_\theta l(\theta, \mathbf{x}_i)\| \|\nabla_\theta l(\theta, \mathbf{x}_j)\|},$$

where l is the loss function and θ is the weight vector. The gradient covariance matrix is strongly related to generalization and interference (Fort et al. 2020, Lyle et al. 2023) — negative off-diagonal entries usually indicate interference between different training samples, while positive off-diagonal entries reflect generalization.

Considering Property 2.3 and Theorem 4.1, when Elephant is applied, we expect the off-diagonal entries of the gradient covariance matrix to be close to zero. From Figure 4.9 to Figure 4.18, we present the heatmaps for training DQN in all 10 Atari tasks with 6 activation functions. Specifically, we set $k = 32$ and estimate the gradient covariance matrices at the midpoint (Frame = 24.8M) and end (Frame = 48.8M) of training. Indeed, as shown in these figures, the gradient covariance matrices exhibits near-zero off-diagonal values throughout the entire training process when Elephant is used, indicating mild generalization and reduced interference. However, for other activation functions such as ReLU, some off-diagonal values remain noticeably non-zero, indicating overgeneralization and strong interference.

Sensitivity Analysis

Finally, we conduct a sensitivity analysis for the hyper-parameters of Elephant. Specifically, we vary σ and d (see Equation (4.3)) in Elephant and test DQN in Battlezone and Q*bert. For reference, we also show the performance of ReLU. As shown in Figure 4.8, for Battlezone, a small a (around



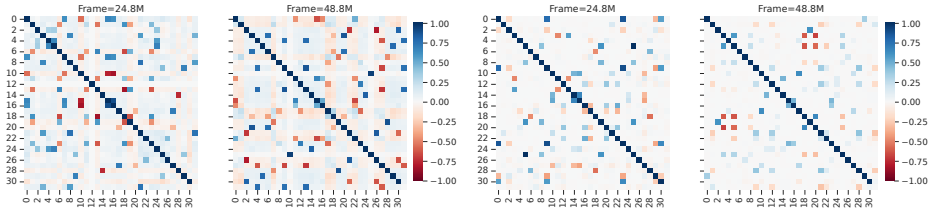
(a) Hyper-parameter a (b) Hyper-parameter d

Figure 4.8: A sensitivity analysis of hyper-parameters σ and d in Elephant. We present the final test returns averaged over 5 runs, and the shaded areas represent standard errors. The best a varies in different tasks while a small d works well across tasks in general.

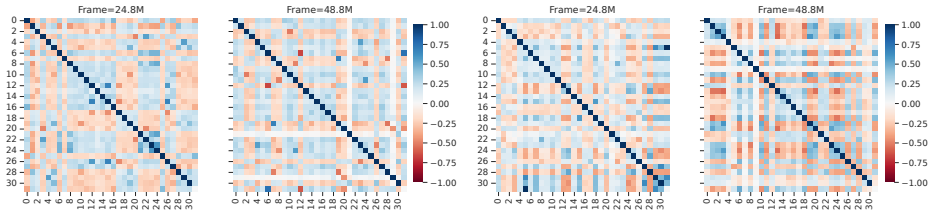
0.1) yields the best performance, whereas for Q*bert, a relatively large a (around 1.2) performs best. As for d, the performance is more robust to d and a small d works well across tasks in general.

4.5 Conclusion

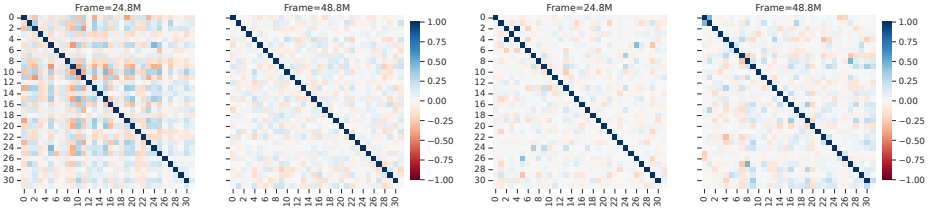
In this chapter, we proposed elephant activation functions that can make neural networks more resilient to catastrophic forgetting. Theoretically, our work provided a deeper understanding of the role of activation functions in catastrophic forgetting. Empirically, we showed that incorporating elephant activation functions in neural networks improves memory efficiency and learning performance of value-based algorithms.



(a) ReLU (b) Tanh

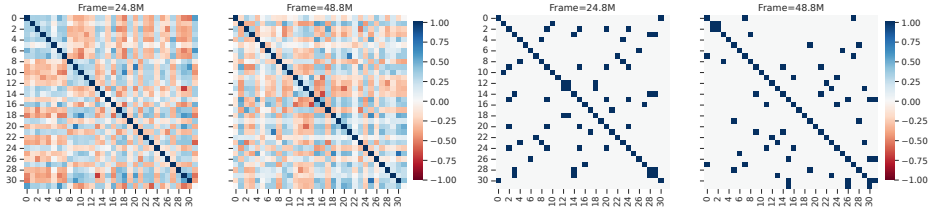


(c) Maxout (d) LWTA

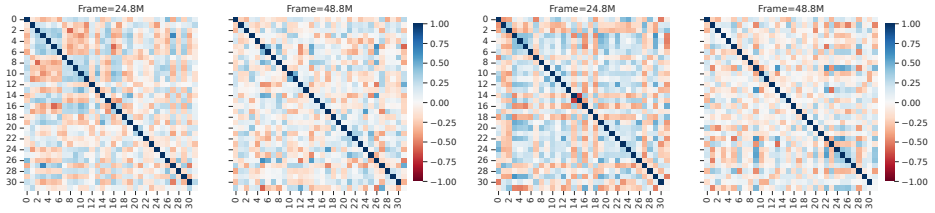


(e) FTA (f) Elephant

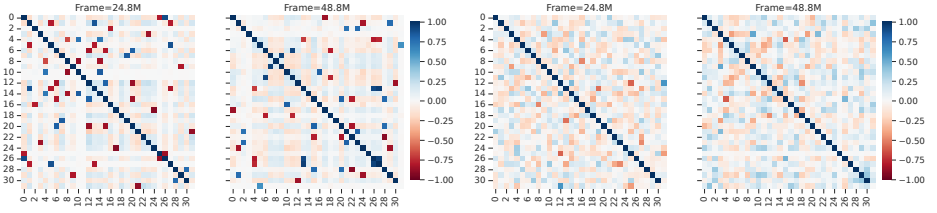
Figure 4.9: Heatmaps of gradient covariance matrices for training DQN in Amidar.



(a) ReLU (b) Tanh

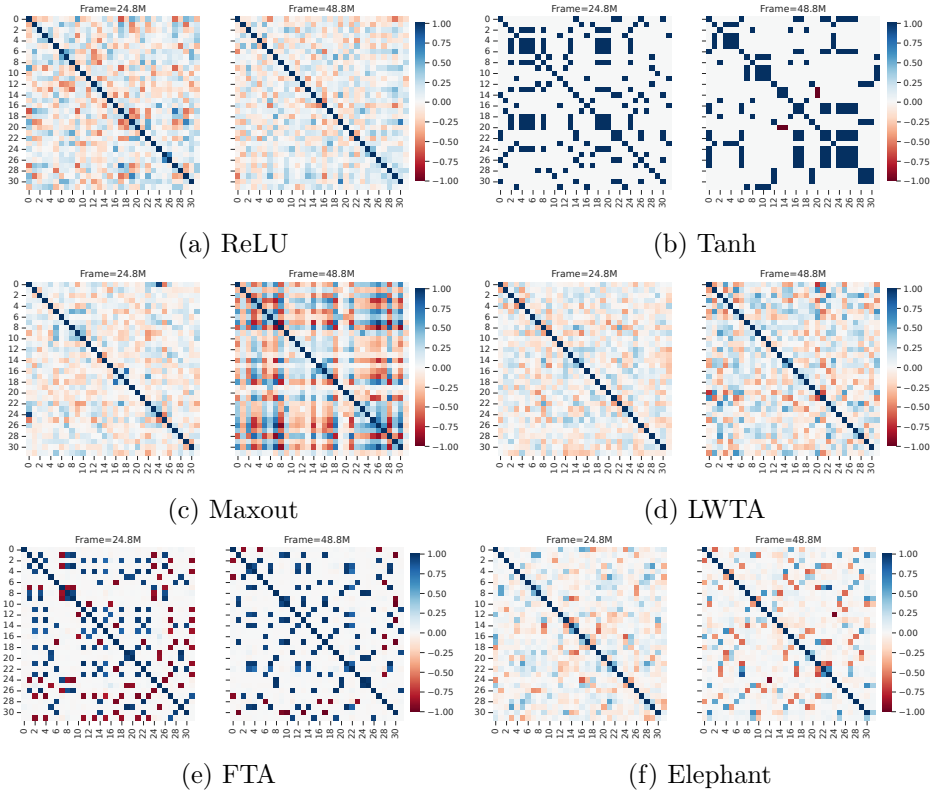


(c) Maxout (d) LWTA



(e) FTA (f) Elephant

Figure 4.10: Heatmaps of gradient covariance matrices for training DQN in Battlezone.

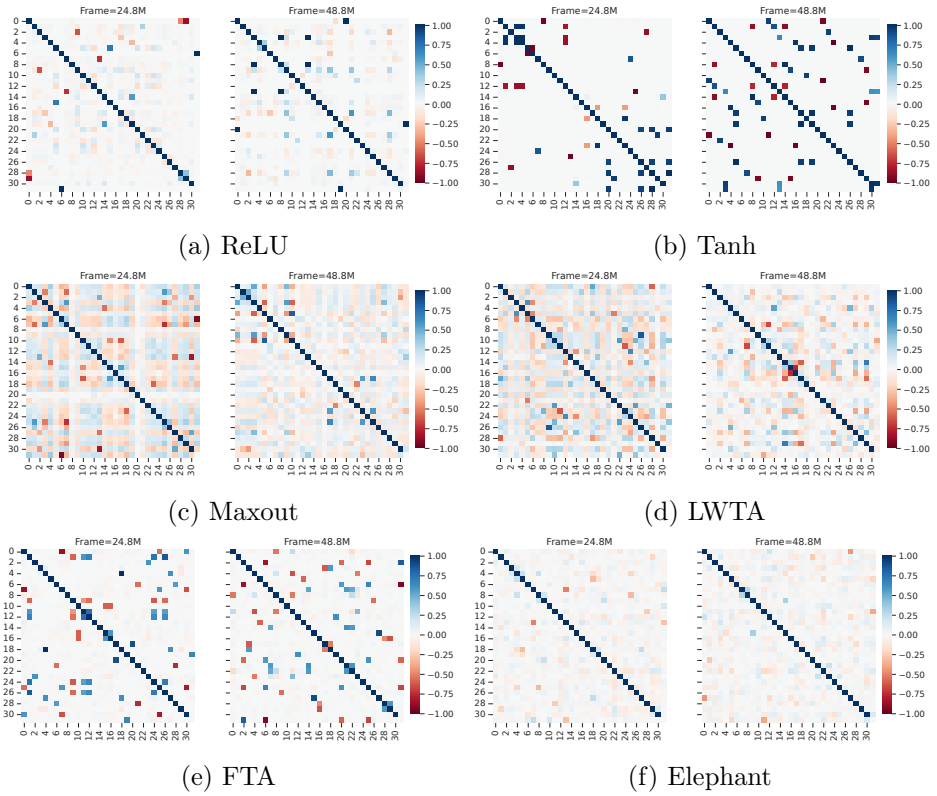


(a) ReLU (b) Tanh

(c) Maxout (d) LWTA

(e) FTA (f) Elephant

Figure 4.11: Heatmaps of gradient covariance matrices for training DQN in Bowling.

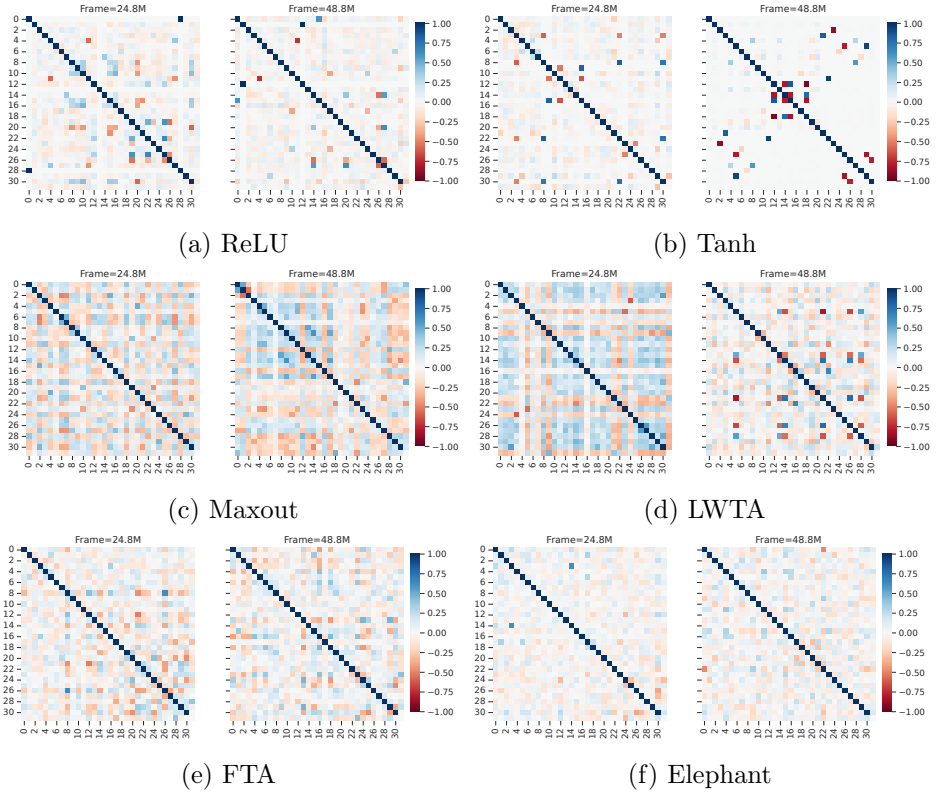


(a) ReLU (b) Tanh

(c) Maxout (d) LWTA

(e) FTA (f) Elephant

Figure 4.12: Heatmaps of gradient covariance matrices for training DQN in Double Dunk.

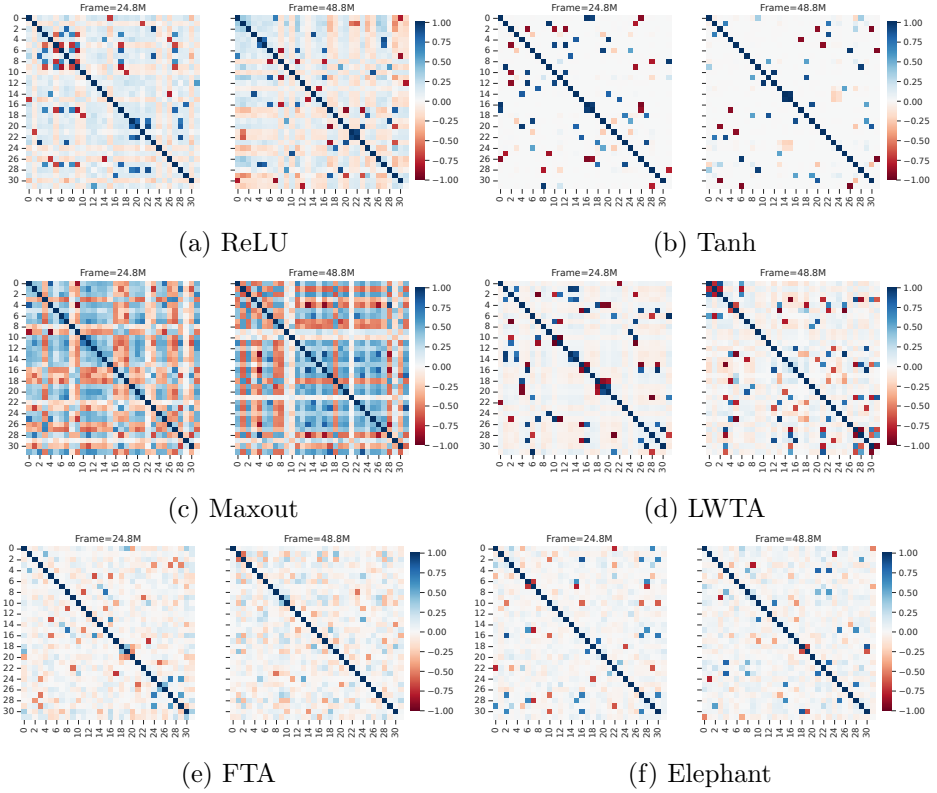


(a) ReLU (b) Tanh

(c) Maxout (d) LWTA

(e) FTA (f) Elephant

Figure 4.13: Heatmaps of gradient covariance matrices for training DQN with in Frostbite.

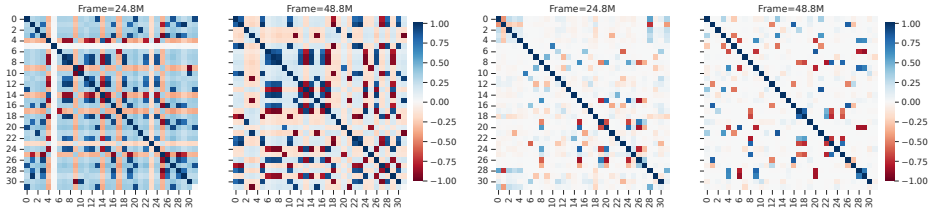


(a) ReLU (b) Tanh

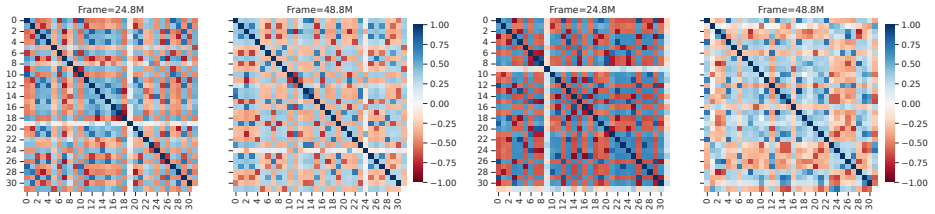
(c) Maxout (d) LWTA

(e) FTA (f) Elephant

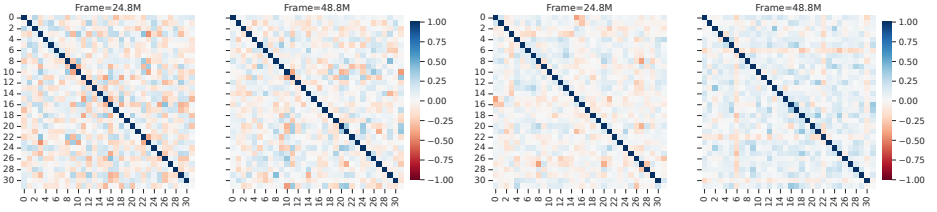
Figure 4.14: Heatmaps of gradient covariance matrices for training DQN in Kung-Fu Master.



(a) ReLU (b) Tanh

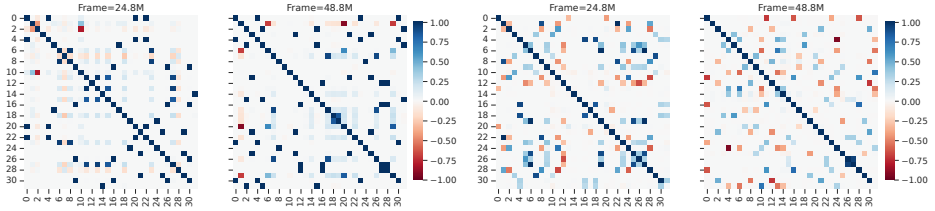


(c) Maxout (d) LWTA

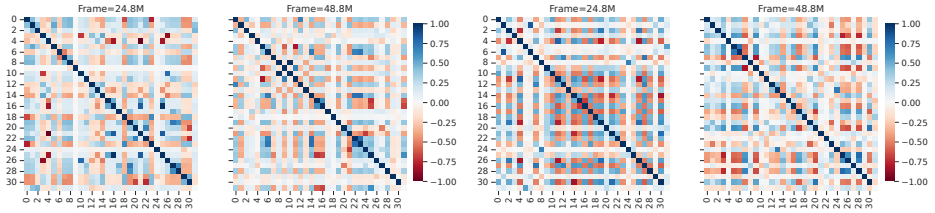


(e) FTA (f) Elephant

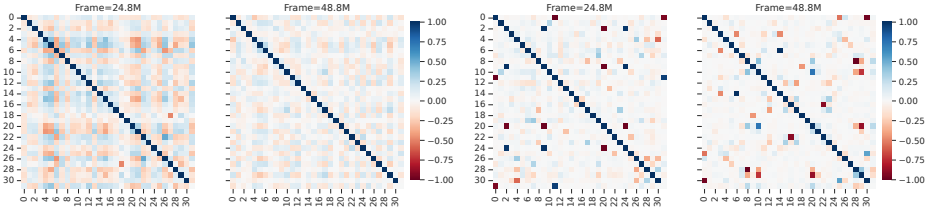
Figure 4.15: Heatmaps of gradient covariance matrices for training DQN in Name This Game.



(a) ReLU (b) Tanh

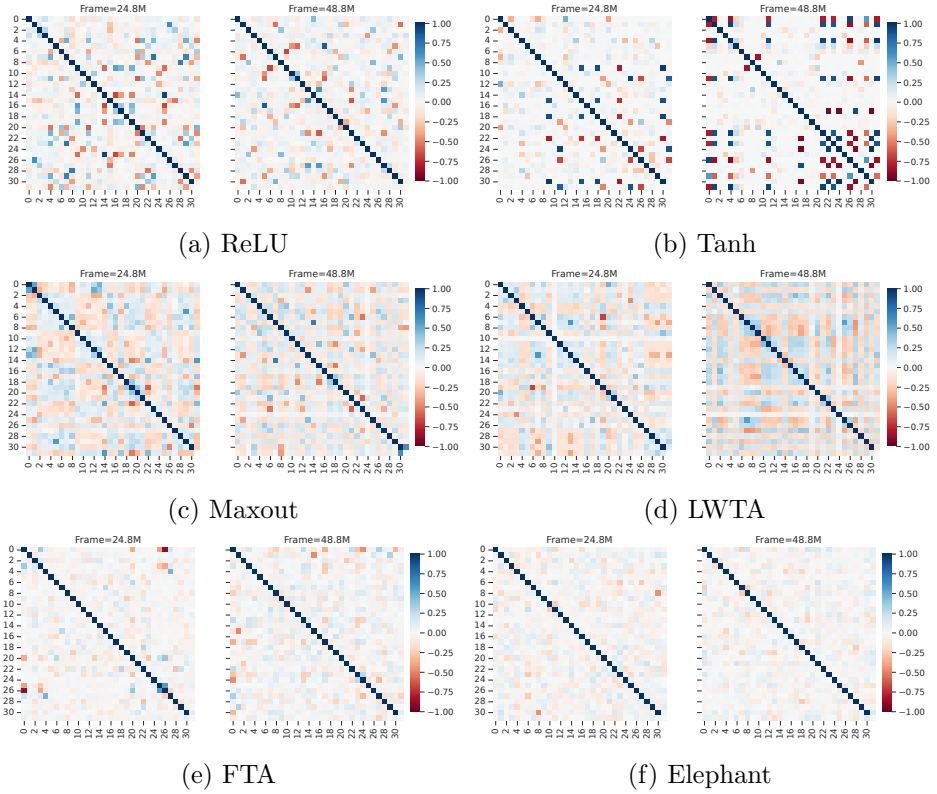


(c) Maxout (d) LWTA



(e) FTA (f) Elephant

Figure 4.16: Heatmaps of gradient covariance matrices for training DQN in Phoenix.

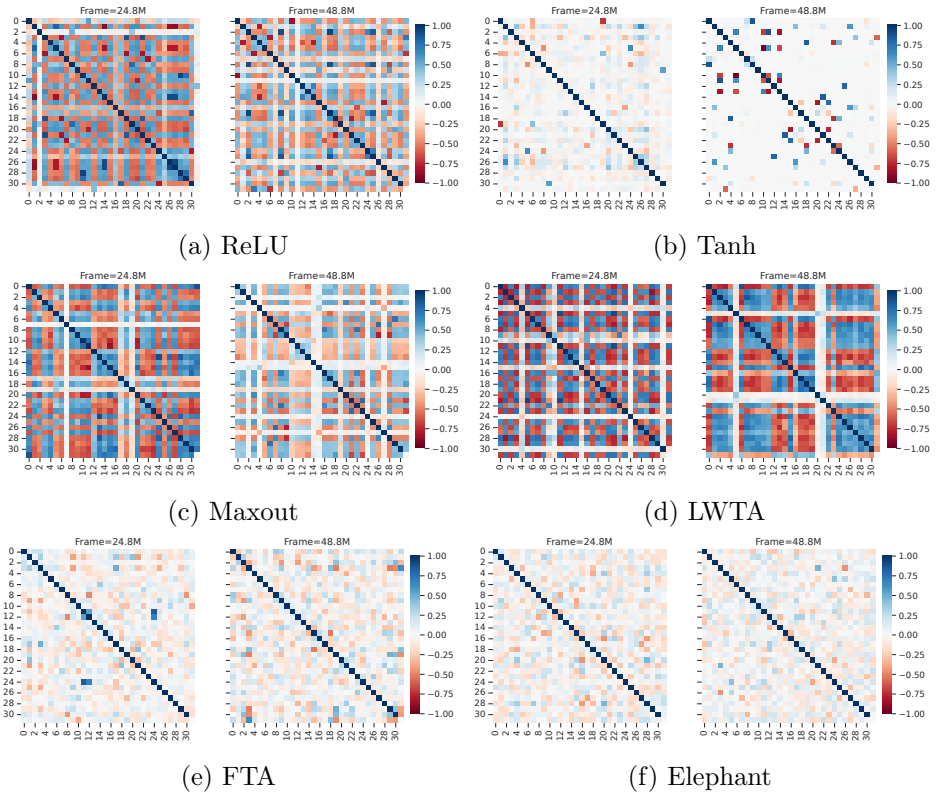


(a) ReLU (b) Tanh

(c) Maxout (d) LWTA

(e) FTA (f) Elephant

Figure 4.17: Heatmaps of gradient covariance matrices for training DQN with in Q*bert.



(a) ReLU (b) Tanh

(c) Maxout (d) LWTA

(e) FTA (f) Elephant

Figure 4.18: Heatmaps of gradient covariance matrices for training DQN with in River Raid.

Chapter 5

Learning to Optimize for Reinforcement Learning

Deep learning has achieved great success in many areas (LeCun et al. 2015), which is largely attributed to the automatically learned features that surpass handcrafted expert features. Gradient descent enables automatic parameter adjustment within a neural network, resulting in highly effective features. Despite these advancements, as another important component in deep learning, optimizers are still largely hand-designed and heavily reliant on expert knowledge. To reduce the burden of hand-designing optimizers, researchers propose to learn to optimize with the help of meta-learning (Sutton 1992a, Andrychowicz et al. 2016, Chen et al. 2017, Wichrowska et al. 2017, Maheswaranathan et al. 2021). Specifically, a learned optimizer is typically parameterized as a neural network which ingests optimization related information (e.g., gradients, losses, or parameter histories) and outputs parameter updates, designed to replace classical hand-crafted optimizers such as SGD, Adam, and RMSProp. Compared to designing optimizers with human expert knowledge, learning an optimizer is a data-driven approach, reducing the reliance on expert knowledge. During training, a learned optimizer can be optimized to speed learning and help achieve better performance. Despite the significant progress in learning optimizers, previous works only present learned optimizers designed for supervised learning (SL). Hoever, they have been shown to perform

poorly in reinforcement learning (RL) tasks (Metz et al. 2020b; 2022b). Learning to optimize for RL remains an open and challenging problem.

One key difficulty is that RL tasks possess unique properties that are largely overlooked by classical optimizers. For instance, the non-stationarity inherent in RL training complicates the optimization of learned optimizers and significantly reduces learning efficiency. Specifically, unlike SL, the input distribution of an RL agent is non-stationary and non-independent and identically distributed (non-IID) due to locally correlated transition dynamics (Alt et al. 2019). Additionally, due to policy and value iterations, the target function and the loss landscapes in RL are constantly changing throughout the learning process, resulting in a much more unstable and complex optimization process. In some cases, these properties also make it inappropriate to apply optimization algorithms designed for SL to RL directly, such as stale accumulated gradients (Bengio et al. 2020a) or unique interference-generalization phenomenon (Bengio et al. 2020b).

In this chapter, we aim to learn optimizers for RL. Instead of manually designing optimizers by studying RL optimization, we apply meta-learning to learn optimizers from data generated in the agent-environment interactions. We first investigate the problem and find that the complicated agent-gradient distribution impedes the training of learned optimizers for RL. Furthermore, the non-IID nature of the agent-gradient distribution also hinders meta-training. Lastly, the highly stochastic agent-environment interactions can lead to agent-gradients with high bias and variance, exacerbating the difficulty of learning an optimizer for RL. In response to these challenges, we propose a novel approach, *Optim4RL*, a learned optimizer for RL that involves pipeline training and a specialized optimizer structure of good inductive bias. Compared with previous methods, Optim4RL is more stable to train and more effective in optimizing RL tasks, without complex optimizer structures or numerous human-designed input features. We demonstrate that Optim4RL can learn to optimize RL tasks from scratch and generalize to unseen tasks. Our work is the first to propose a learned optimizer for deep RL tasks that works well in practice.

5.1 Learning to Optimize with Meta-Learning

We aim to learn an optimizer with meta-gradient methods. Let θ be the agent-parameters of an RL agent that we aim to optimize. A (learned) optimizer is defined as an update function U that maps input gradients to parameter updates, implemented as a meta-network, parameterized by the meta-parameters ϕ. Let z be the input of this meta-network which may include gradients g, losses L, exponential moving average of gradients, etc. Let h be an optimizer state which stores historical values. We can then compute agent-parameters updates $\Delta\theta$ and the updated agent-parameters θ':

$$\Delta\theta, h' = U_\phi(z, h) \text{ and } \theta' = \theta + \Delta\theta.$$

Note that all classical first-order optimizers can be written in this form with $\phi = \emptyset$. As an illustration, for SGD, $h' = h = \emptyset$, $z = g$, and $U_{\text{SGD}}(g, \emptyset) = (-\alpha g, \emptyset)$, where α is the learning rate. For RMSProp (Tieleman and Hinton 2012), set $z = g$; h is used to store the average of squared gradients. Then $U_{\text{RMSProp}}(g, h) = (-\frac{\alpha g}{\sqrt{h'+\epsilon}}, h')$, where $h' = \beta h + (1-\beta)g^2$, $\beta \in [0, 1]$, and ϵ is a tiny positive number for numerical stability.

Similar to Xu et al. (2020), we apply bilevel optimization to optimize θ and ϕ. First, we collect $M+1$ trajectories $\mathcal{T} = \{\tau_0, \tau_1, \cdots, \tau_{M-1}, \tau_M\}$. For the inner update, we fix ϕ and apply multiple steps of gradient descent updates to θ by minimizing an inner loss L^{inner}. Specifically, for each trajectory $\tau_i \in \mathcal{T}$, we have

$$\Delta\theta_i \propto \nabla_\theta L^{\text{inner}}(\tau_i; \theta_i, \phi) \text{ and } \theta_{i+1} = \theta_i + \Delta\theta_i,$$

where $\nabla_\theta L^{\text{inner}}$ are agent-gradients of θ. By repeating the above process for M times, we get $\theta_0 \xrightarrow{\phi} \theta_1 \cdots \xrightarrow{\phi} \theta_M$. Here, θ_M are functions of ϕ. For simplicity, we abuse the notation and still use θ_M. Next, we use τ_M as a validation trajectory to optimize ϕ with an outer loss L^{outer}:

$$\Delta\phi \propto \nabla_\phi L^{\text{outer}}(\tau_M; \theta_M, \phi) \text{ and } \phi' = \phi + \Delta\phi,$$

where $\nabla_\phi L^{\text{outer}}$ are meta-gradients of ϕ. Since θ_M are functions of ϕ, we can apply the chain rule to compute meta-gradients $\nabla_\phi L^{\text{outer}}$. Next, we reset $\theta_0 = \theta_M$ and repeat the above bilevel optimization process until the end of training.

In our experiments, we use JAX (Bradbury et al. 2018) to perform reverse-mode automatic differentiation. Specifically, we apply the `jax.grad()` function to compute the agent-gradients $\nabla_\theta L^{\text{inner}}$. For meta-gradients $\nabla_\phi L^{\text{outer}}$, which involve higher-order derivatives, JAX's autodiff enables straightforward computation by simply stacking `jax.grad()`, as the functions that compute derivatives are themselves differentiable.

5.2 Related Work

Our work is closely related to two areas: optimization in RL and learning to optimize in SL.

5.2.1 Optimization in Reinforcement Learning

In practice, temporal-difference (TD) is widely applied for value iterations in many RL algorithms:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)), \tag{5.1}$$

where α is the learning rate, s_t and s_{t+1} are two successive states, and $r_{t+1} + \gamma V(s_{t+1})$ is named the TD target. TD targets are usually biased, non-stationary, and noisy due to changing state-values, complex state transitions, and noisy reward signals (Schulman et al. 2016). They usually induce a changing loss landscape that evolves during training. As a result, the agent-gradients usually have high bias and variance which can lead to sub-optimal performance or even a failure of convergence.[1]

Henderson et al. (2018) tested different optimizers in RL and pointed out that classical adaptive optimizers may not always consider the complex interactions between RL algorithms and environments. Sarigül and Avci (2018) benchmarked different momentum strategies in deep RL and found

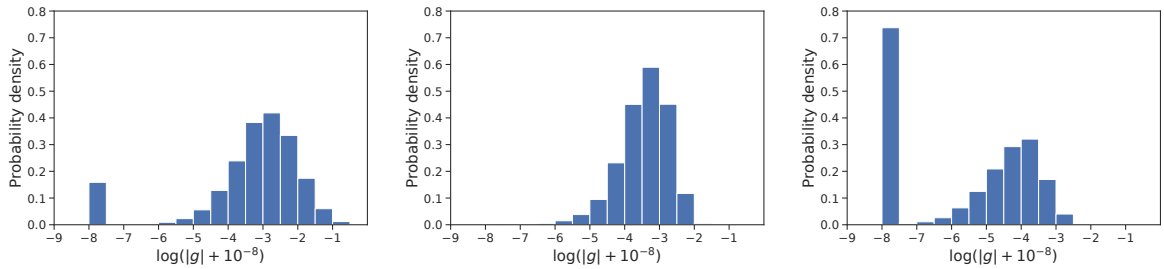
[1] In this chapter, we use the term agent-gradients to refer to gradients of all parameters in a learning agent, which may include policy gradient, gradient of value functions, gradient of other hyper-parameters.

that Nesterov momentum is better at generalization. Bengio et al. (2020a) took one step further and showed that unlike SL, momentum in TD learning becomes doubly stale due to changing parameter updates and bootstrapping. By correcting momentum in TD learning, the sample efficiency can be improved. These works together indicate that it may not always be appropriate to bring optimization methods in SL directly to RL without considering the unique properties in RL. Unlike these works which hand-design new optimizers for RL, we adopt a data-driven approach and apply meta-learning to learn an RL optimizer from data generated in the agent-environment interactions.

5.2.2 Learning to Optimize in Supervised Learning

Initially, learning to optimize is only applied to tune the learning rate (Jacobs 1988, Sutton 1992a, Mahmood et al. 2012). Recently, researchers started to learn an optimizer completely from scratch, i.e., not just tune the learning rate but learn the entire weight update function. Andrychowicz et al. (2016) implemented learned optimizers with long short-term memory networks (Hochreiter and Schmidhuber 1997) and showed that learned optimizers could generalize to unseen tasks. Li and Malik (2017) applied a guided policy search method to find a good optimizer. Wichrowska et al. (2017) introduced a hierarchical recurrent neural network (RNN) (Medsker and Jain 2001) architecture, which greatly reduces memory and computation, and was shown to generalize to different network structures. Metz et al. (2022a) developed learned optimizers with multi-layer perceptions, which achieve a better balance among memory, computation, and performance.

Training learned optimizers from scratch are known to be hard. Part of the reason is that they are usually trained by truncated backpropagation through time, which leads to strongly biased gradients or exploding gradients. To overcome these issues, Metz et al. (2019) presented a method to dynamically weigh a reparameterization gradient estimator and an evolutionary strategy style gradient estimator, stabilizing the training of learned optimizers. Vicol et al. (2021) resolved the issues by dividing the computation graph into truncated unrolls and computing unbiased gradients with evolution strategies and gradient bias corrections. Harrison et al. (2022) investigated the training stability of optimization algorithms and proposed to improve the stability of learned optimizers by



(a) At the beginning of training (b) In the middle of training (c) At the end of training

Figure 5.1: A visualization of agent-gradient distributions at different training stages, showing that the agent-gradient distribution is non-IID during training. All agent-gradients are collected during training A2C in big_dense_long, optimized by RMSProp. We compute $\log(|g| + 10^{-8})$ to avoid the error of applying log function to non-positive agent-gradients.

adding adaptive nominal terms from Adam (Kingma and Ba 2015) and AggMo (Lucas et al. 2019). Metz et al. (2020a) trained a general-purpose optimizer by training optimizers on thousands of tasks with a large amount of computation. Following the same spirit, Metz et al. (2022b) continued to perform large-scale optimizer training, leveraging more computation ($4,000$ TPU-months) and more diverse SL tasks. The learned optimizer, VeLO, requires no hyperparameter tuning and works well on a wide range of SL tasks. VeLO is the precious outcome of long-time research in the area of learning to optimize, building on the wisdom and effort of many generations. Although marking a milestone for the success of learned optimizers in SL tasks, VeLO still performs poorly in RL tasks, as shown in Section 4.4.4 in Metz et al. (2022b).

The failure of VeLO in RL tasks suggests that designing learned optimizers for RL is still a challenging problem. Unlike previous works that focus on learning optimizers for SL, we aim to learn to optimize for RL. As we will show next, our method is simple, stable, and effective, without using complex network structures or incorporating numerous human-designed features. As far as we know, our work is the first to demonstrate the success of learned optimizers in deep RL tasks.

5.3 Issues in Learning to Optimize for Reinforcement Learning

Learned optimizers for SL are infamously hard to train, suffering from high training instability (Wichrowska et al. 2017, Metz et al. 2019; 2020a). Learning an optimizer for RL is even harder (Metz et al. 2022b). In the sections that follow, we identify two important issues in learning to optimize for RL.

5.3.1 The Agent-Gradient Distribution is Non-IID

In RL, a learned optimizer takes the agent-gradient g as an input and outputs the agent-parameter update $\Delta\theta$. To investigate the hardness of learning an optimizer for RL, we collect agent-gradients by training A2C (Mnih et al. 2016) in big_dense_long (see Section 5.5 for details) for $30M$ steps with learning rate $3e-3$, optimized by RMSProp (Tieleman and Hinton 2012). All collected agent-gradients are divided into 30 parts by time-steps. We then plot the agent-gradients in the first, sixteenth, and last parts as the agent-gradient distributions at the beginning, middle, and end of training, respectively. The plotted agent-gradients are shown with logarithmic x-axis in Figure 5.1. The y-axis shows the probability density. Clearly, the agent-gradient distribution is non-IID, changing throughout the training process. Specifically, at the beginning of training, there are two peaks in the agent-gradient distribution. In the middle of training, most agent-gradients are non-zero, concentrated around 10^{-3}. At the end of the training, a large portion of the agent-gradients are zeros. It is well-known that a non-IID input distribution makes training more unstable and reduces learning performance in many settings (Ma et al. 2022, Wang et al. 2024, Khetarpal et al. 2022). Similarly, the violation of the IID assumption would also increase learning instability and decrease efficiency for training learned optimizers. Note that this issue exists in both learning to optimize for SL and RL. However, the agent-gradient distribution from RL is generally more non-IID than the gradient distribution from SL, since RL tasks are inherently more non-stationary.

5.3.2 A Vicious Spiral of Bilevel Optimization

Learning an optimizer while optimizing parameters of a model is a bilevel optimization, suffering from high training instability (Wichrowska et al. 2017, Metz et al. 2020a, Harrison et al. 2022). In RL, due to highly stochastic agent-environment interactions, the agent-gradients have high bias and variance, which make the bilevel optimization even more unstable.

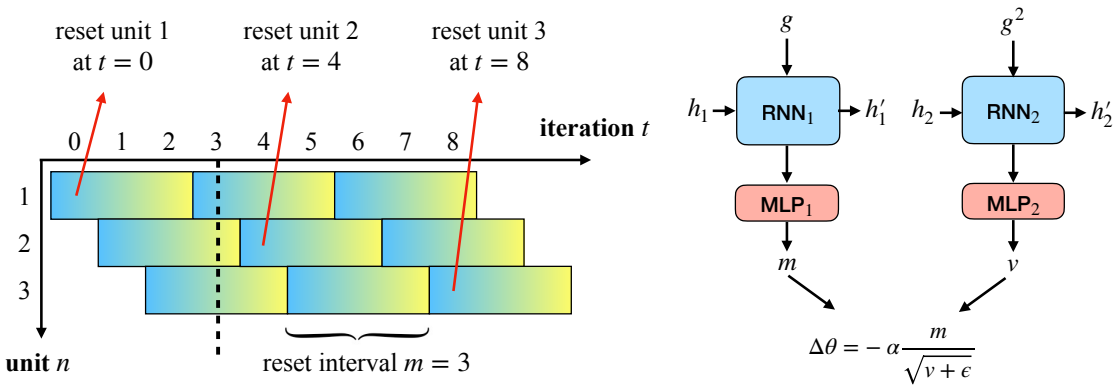
Specifically, in SL, it is often assumed that the training set consists of IID samples. However, the input data distribution in RL is non-IID, which makes the whole training process much more unstable and complex, especially when learning to optimize is involved. In most SL settings, true labels are noiseless and time-invariant. For example, the true label of a written digit 2 in MNIST (Deng 2012) is $y = 2$, which does not change during training. In RL, TD learning (see Equation (5.1)) is widely used, and TD targets play a similar role as labels in SL. Unlike labels in SL, TD targets are biased, non-stationary, and noisy, due to highly stochastic agent-environment interactions. This leads to a loss landscape that evolves during training and potentially results in the deadly triad (Van Hasselt et al. 2018) and capacity loss (Lyle et al. 2021). Moreover, in SL, a lower loss usually indicates better performance (e.g., higher classification accuracy). But in RL, a lower outer loss is not necessarily a good indicator of better performance (i.e., higher return) due to a changing loss landscape. Together with biased TD targets, the randomness from state transitions, reward signals, and agent-environment interactions, make the bias and variance of agent-gradients relatively high. In learning to optimize for RL, meta-gradients are afflicted with large noise induced by the high bias and variance of agent-gradients. With noisy and inaccurate meta-gradients, the improvement of the learned optimizer (i.e., the outer update) is unstable and slow. Using a poorly performed optimizer, policy improvement (i.e., the inner update) is no longer guaranteed. A poorly performed agent is unlikely to collect "high-quality" data to boost the performance of the agent and the learned optimizer. In the end, this bilevel optimization gets stuck in a vicious spiral: a poor optimizer $\rightarrow$ a poor agent policy $\rightarrow$ collected data of low-quality $\rightarrow$ a poor optimizer $\rightarrow \cdots$.

5.4 Optim4RL: A Learned Optimizer for Reinforcement Learning

To overcome the issues in Section 5.3, we propose a learned optimizer for RL, named *Optim4RL*, which incorporates pipeline training and a novel optimizer structure. As we will show next, Optim4RL is more robust and efficient to train than previous methods.

5.4.1 Pipeline Training

In Figure 5.1, we show that the agent-gradient distribution is non-IID during training. Generally, a good optimizer should be well-functioned under different agent-gradient distributions in the whole training process. To make the agent-gradient distribution more IID, we propose *pipeline training*.



(a) Pipeline training

(b) The optimizer structure of Optim4RL

Figure 5.2: (a) An example of pipeline training where the reset interval $m = 3$ and the number of units $n = 3$. All training units are reset at regular intervals to diversify training data. (b) The network structure of Optim4RL. g is the input agent-gradient, h_i and h_i' are hidden states, α is the learning rate, ϵ is a small positive constant, and $\Delta\theta$ is the parameter update.

Instead of training only one agent, we train n agents in parallel, each with its own task and optimizer state, following the common practice of learned optimizers. Together, the three elements form a *training unit* (agent, task, optimizer state); and we have n training units in total. Let m be a positive integer we call the *reset interval*. A complete *training interval* lasts for m training iterations. In Figure 5.2 (a), we show an example of pipeline training with $m = n = 3$. To train an optimizer effectively, the input of the learned optimizer includes agent-gradients from all n training

units. Before training, we choose n integers $\{r_1, \cdots, r_n\}$ such that they are evenly spaced over the interval $[0, m-1]$. Then we assign r_i to training unit i for $i \in \{1, \cdots, n\}$. We also apply a delayed start strategy by setting the starting iteration of training unit i as iteration $i-1$. At training iteration t, we reset training unit i if $r_i \equiv t \pmod{m}$. By resetting training units at regular intervals and using the delayed start strategy, it is guaranteed that for most iterations, we can access training data across one training interval. For instance, at $t = 3$, the input consists of agent-gradients from unit 1 at the beginning of an interval, agent-gradients from unit 2 at the end of an interval, and agent-gradients from unit 3 in the middle of an interval, indicated by the dashed line in Figure 5.2 (a). With pipeline training, the input agent-gradients are more diverse and spread across a whole training interval, making the input distribution more IID. Ideally, we expect $m \leq n$ so that the input consists of agent-gradients from all training stages. In our experiments, n is the number of training environments; m depends on the training steps of each task, and it has a similar magnitude as n.

5.4.2 Improving the Inductive Bias of Learned Optimizers

Recently, Harrison et al. (2022) proved that adding adaptive terms to learned optimizers improves the training stability of optimizing a noisy quadratic model. Experimentally, Harrison et al. (2022) showed that adding terms from Adam (Kingma and Ba 2015) and AggMo (Lucas et al. 2019) improves the stability of learned optimizers as well. However, including human-designed features not only makes an optimizer more complex but is also against the spirit of learning to optimize—ideal learned optimizers should be able to automatically learn useful features, reducing the reliance on human expert knowledge as much as possible. Instead of incorporating terms from adaptive optimizers directly, we design the parameter update function in a similar form to adaptive optimizers:

$$\Delta\theta = -\alpha \frac{m}{\sqrt{v + \epsilon}}, \tag{5.2}$$

where α is the learning rate, ϵ is a small positive number, and m and v are the processed outputs of dual-RNNs, as shown in Figure 5.2 (b). Specifically, for each input gradient g, we generate two

Algorithm 5 A Learned Optimizer for Reinforcement Learning (Optim4RL)

Require: RNN_1 and RNN_2, MLP_1 and MLP_2, hidden states h_1 and h_2, input gradient g, $\epsilon = 10^{-8}$, learning rate α.

1: $g \leftarrow\perp g$ ▷ $\perp$ denotes the stop-gradient operation
2: $h_1, x_1 \leftarrow \text{RNN}_1(h_1, g)$ and $o_1 = \text{MLP}_1(x_1)$
3: $m = \text{sign}(g)\exp(o_1)$ ▷ Compute m: 1st pseudo moment estimate
4: $h_2, x_2 \leftarrow \text{RNN}_2(h_2, g^2)$ and $o_2 = \text{MLP}_2(x_2)$
5: $v = \exp(o_2)$ ▷ Compute v: 2nd pseudo moment estimate
6: $\Delta\theta \leftarrow -\alpha\frac{m}{\sqrt{v+\epsilon}}$ ▷ Compute the parameter update

scalars o_1 and o_2. We then set $m = g_{sign}\exp(o_1)$ and $v = \exp(o_2)$, where $g_{sign} \in \{-1, 1\}$ is the sign of g. More details are included in Algorithm 5.

By parameterizing the parameter update function as Equation (5.2), we improve the inductive bias of learned optimizers by choosing a suitable hypothesis space for learned optimizers and reducing the burden of approximating square root and division for neural networks. In general, we want to learn a good optimizer in a reasonable hypothesis space. It should be large enough to include as many good optimizers as possible, such as Adam (Kingma and Ba 2015) and RMSProp (Tieleman and Hinton 2012). Meanwhile, it should also rule out bad choices so that a suitable candidate can be found efficiently. An optimizer in the form of Equation (5.2) meets the two requirements exactly. Moreover, it is generally hard for neural networks to approximate mathematical operations accurately (Telgarsky 2017, Yarotsky 2017, Boullé et al. 2020, Lu et al. 2021). With Equation (5.2), a neural network can spend all its expressivity and capacity learning m and v, reducing the burden of approximating square root and division.

Finally, we combine the two techniques and propose our method—a learned optimizer for RL (Optim4RL). Following Andrychowicz et al. (2016), our optimizer also operates coordinatewisely on agent-parameters so that all agent-parameters share the same optimizer. Besides gradients, many previously learned optimizers for SL include human-designed features as inputs, such as moving average of gradient values at multiple timescales, moving average of squared gradients, and Adafactor-style accumulators (Shazeer and Stern 2018). In theory, these features can be learned and stored in the hidden states of RNNs in Optim4RL. So for simplicity, we only consider agent-gradients as inputs. As we will show next, despite its simplicity, our learned optimizer Optim4RL

achieves satisfactory performance in many RL tasks, outperforming several learned optimizers.

5.5 Experiments

In this section, we first verify that Optim4RL can learn to optimize for RL from scratch. Then, we show how to train a general-purpose learned optimizer for RL. Finally, we demonstrate that Optim4RL enjoys the advantage of robust training and achieves strong generalization under different hyper-parameter settings. Note that in all our experiments, we use JAX (Bradbury et al. 2018) for automatic differentiation. To ensure compatibility, all algorithms and tasks in this section are also implemented with JAX.

Tasks Following Oh et al. (2020), we design 6 gridworlds with various properties, such as different horizons, reward functions, or state-action spaces. In each gridworld, there are N objects. Each object is described as $[r, \epsilon_{\text{term}}, \epsilon_{\text{respawn}}]$. Object locations are randomly determined at the beginning of each episode, and an object reappears at a random location after being collected, with a probability of $\epsilon_{\text{respawn}}$ for each time-step. The observation consists of a tensor $\{0,1\}^{N\times H\times W}$, where N is the number of objects, and $H \times W$ is the size of the grid. An agent has 9 movement actions for adjacent positions, including staying in the same position. When the agent collects an object, it receives the corresponding reward ($r \times$ reward scale), and the episode terminates with a probability of ϵ_{term} associated with the object. The default reward scale is 1. In Table 5.1, we describe the setting of each gridworld in detail.

Besides gridworlds, we also test our method in Catch (Osband et al. 2020) and Brax tasks (Freeman et al. 2021). Note that we use Brax tasks instead of the more widely adopted MuJoCo tasks (Todorov et al. 2012), as MuJoCo is not compatible with JAX. Specifically, Brax tasks are implemented in JAX and are designed to closely resemble MuJoCo control tasks.

RL Algorithms and Training Settings We mainly consider two RL algorithms—A2C (Mnih et al. 2016) and PPO (Schulman et al. 2017). For all experiments, we train A2C in gridworlds

Table 5.1: The detailed settings of gridworlds.

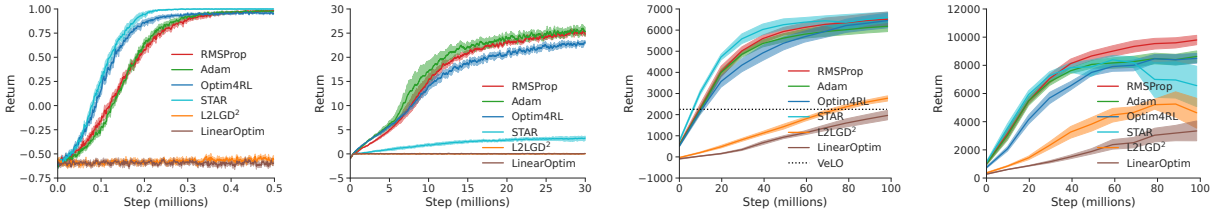
Task \ Setting	Size ($H \times W$)	Objects	Horizon
big_sparse_short	10×12	$2 \times [1.0, 0.0, 0.05]$, $2 \times [-1.0, 0.5, 0.05]$	50
big_sparse_long	12×10	$2 \times [1.0, 0.0, 0.05]$, $2 \times [-1.0, 0.5, 0.05]$	500
big_dense_short	9×13	$2 \times [1.0, 0.0, 0.5]$, $2 \times [-1.0, 0.5, 0.5]$	50
big_dense_long	13×9	$2 \times [1.0, 0.0, 0.5]$, $2 \times [-1.0, 0.5, 0.5]$	500
small_dense_long	6×4	$[1.0, 0.0, 0.5]$, $[-1.0, 0.5, 0.5]$	500
small_dense_short	4×6	$[1.0, 0.0, 0.5]$, $[-1.0, 0.5, 0.5]$	50

and train PPO in Brax tasks. For A2C training in gridworlds, the feature net is an MLP with hidden size 32 for the "small" gridworlds. For the "big" gridworlds, the feature net is a convolution neural network (CNN) with 16 features and kernel size 2, followed by an MLP with output size 32. Unless mentioned explicitly, we use ReLU as the activation function. We set $\lambda = 0.95$ to compute λ-returns. The discount factor $\gamma = 0.995$. One rollout has 20 steps. The actor loss weight is 1.0, the critic loss weight is 0.5, and the entropy weight is 0.01. The final inner loss is defined as a weighted sum of the actor loss, the critic loss, and the entropy loss.

For PPO training in Brax games, we use the same settings in Brax examples.[2] Specifically, the actor loss weight is 1.0, the critic loss weight is 0.5, and the entropy weight is 0.01 (Ant and Pendulum) or 0.001 (Humanoid and Walker2D). Similar to A2C, the final inner loss is also a weighted sum of the actor loss, the critic loss, and the entropy loss.

For Optim4RL, due to resource constraints, we choose a small network with two gated recurrent unit (GRUs) (Cho et al. 2014) of hidden size 8; both multi-layer perceptrons (MLPs) have two hidden layers of size 16. We use Adam to optimize the learned optimizers, with the outer losses set identical to the corresponding inner losses. To meta-learn optimizers, we set $M = 4$ in all experiments; that is, for every outer update, we do 4 inner updates. Potentially, a larger M could lead to more farsighted learning but results in increasing memory and computation requirements. We set $M = 4$ as a trade-off, which works well in practice. Following common practice (Lu et al. 2022), we report results averaged over 10 runs.

[2] https://github.com/google/brax/blob/main/notebooks/training.ipynb



(a) Catch (b) big_dense_long (c) Ant (d) Humanoid

Figure 5.3: The optimization performance of different optimizers in four RL tasks. Note that the performance of VeLO is estimated based on Figure 11 (a) in Metz et al. (2022b). All other results are averaged over 10 runs, and the shaded areas represent 90% confidence intervals. Optim4RL is the only learned optimizer that achieves satisfactory performance in all tasks.

5.5.1 Learning an Optimizer for RL from Scratch

We first show that it is feasible to train Optim4RL in RL tasks from scratch, while learned optimizers for SL do not work well consistently in RL tasks. We consider both classical (Adam and RMSProp) and learned optimizers (L2LGD^2 (Andrychowicz et al. 2016), STAR (Harrison et al. 2022), and VeLO (Metz et al. 2022b)) as baselines. Except for VeLO, we meta-learn optimizers in one task and then test the *fixed* learned optimizers in this specific task.

For L2LGD^2, the model consists of a GRU with hidden size 8, followed by an MLP with hidden sizes $[16, 16]$. For STAR, we use the official implementation from learned_optimization.[3] Unlike the supervised learning setting, we set weight decay to 0 since a positive weight decay in STAR leads to much worse performance. For a fair comparison, we apply pipeline training to train all learned optimizers. For Catch, the agent learning rate is $1e{-}3$; the number of environments (i.e., the number of training units n) is 64; the reset interval m is chosen from $\{32, 64\}$. For big_dense_long , the agent learning rate is $3e-3$; the number of environments is 512; the reset interval m is chosen from $\{72, 144, 288, 576\}$. For Ant and Humanoid, the agent learning rate is $3e-4$; the number of environments is 2048; the reset interval m is chosen from $\{32, 64, 128, 256, 512\}$. Furthermore, in order to reduce memory requirement, we set the number of mini-batches to 8; and change the hidden sizes of the value network from $[256, 256, 256, 256, 256]$ to $[64, 64, 64, 64, 64]$. We use Adam as the meta optimizer and choose the meta learning rate from $\{1e-5, 3e-5, 1e-4, 3e-4, 1e-$

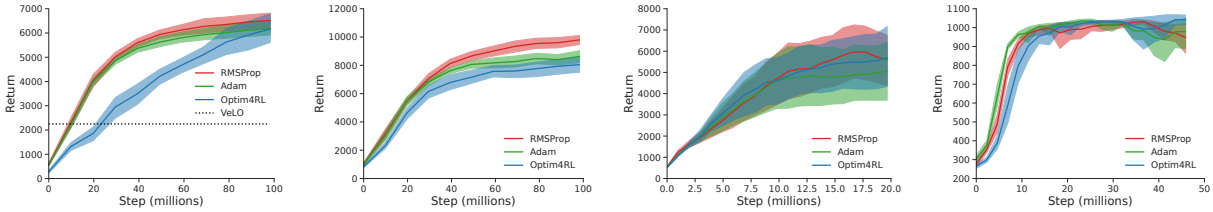
[3] https://github.com/google/learned_optimization/blob/main/learned_optimization/learned_optimizers/adafac_nominal.py

$3, 3e-3, 1e-2\}$.

The optimization performance of optimizers is measured by returns averaging over 10 runs, as shown in Figure 5.3. In general, L2LGD^2 fails in all four tasks. In Catch, both STAR and Optim4RL perform better than classical optimizers (Adam and RMSProp), achieving a faster convergence rate. In Ant, Optim4RL and STAR perform pretty well, on par with Adam and RMSProp, while significantly outperforming the state-of-the-art optimizer—VeLO. However, STAR fails to optimize effectively in big_dense_long; in Humanoid, STAR's performance is unstable and crashes in the end. Optim4RL is the only learned optimizer that achieves stable and satisfactory performance in all tasks, which is a significant accomplishment in its own right, as it demonstrates the efficacy of our approach and its potential for practical applications.

The Advantage of the Inductive Bias of Optim4RL As an ablation study, we demonstrate the advantage of the inductive bias of Optim4RL by comparing it with *LinearOptim*, which has a "linear" parameter update function: $\Delta\theta = -\alpha(a*g+b)$, where α is the learning rate, a and b are the outputs of an RNN model. Specifically, the model of LinearOptim consists of a GRU with hidden size 8, followed by an MLP with hidden sizes $[16, 16]$. The only difference between LinearOptim and Optim4RL is the inductive bias—the parameter update function of LinearOptim is in the form of a linear function. In contrast, the parameter update function of Optim4RL is inspired by adaptive optimizers (see Equation (5.2)). As shown in Figure 5.3, LinearOptim fails to optimize in all tasks, verifying the advantage of the inductive bias of Optim4RL.

The Effectiveness of Pipeline Training By making the input agent-gradient distribution more IID and less time-dependent, pipeline training could improve the training stability and efficiency. To verify this claim, we compare the optimization performance of Optim4RL with and without pipeline training in Table 5.2. We observe minor performance improvement in two gridworlds (small_dense_long and big_dense_long) and more significant improvement in two Brax tasks (Ant and Humanoid), confirming the effectiveness of pipeline training.



(a) Ant (b) Humanoid (c) Pendulum (d) Walker2D

Figure 5.4: Optim4RL shows strong generalization ability and achieves good performance in Brax tasks, although it is only trained in six simple gridworlds from scratch. For comparison, VeLO (Metz et al. 2022b) is trained for 4,000 TPU-months with thousands of tasks but only achieves sub-optimal performance in Ant.

Table 5.2: The performance of Optim4RL with and without pipeline training. All results are averaged over 10 runs, reported with 90% confidence intervals. In general, pipeline training helps improve performance.

Method \ Task	small_dense_long	big_dense_long	Ant	Humanoid
With Pipeline Training	32.22±0.52	23.10±0.31	6421±355	8440±364
W.o. Pipeline Training	30.64±0.69	22.47±0.51	5038±235	6557±1055

5.5.2 Toward a General-Purpose Learned Optimizer for RL

A general-purpose optimizer should perform well even when the input gradients are at various scales. To meta-train a learned general-purpose optimizer, first we design six gridworlds such that the generated agent-gradients in these tasks vary across a wide range. To demonstrate the generalization ability of Optim4RL, we then meta-train Optim4RL in these gridworlds with A2C and test it in Brax tasks with PPO.

Specifically, we use Adam as the meta optimizer and choose the meta learning rate from $\{1e-5, 3e-5, 1e-4, 3e-4, 1e-3, 3e-3\}$. The number of environments/training units n is 512. The reset interval m is chosen from $\{72, 144, 288, 576\}$. The reward scales of all gridworlds are in Table 5.3.

As shown in Figure 5.4, Optim4RL achieves satisfactory performance in these tasks, showing a strong generalization ability. Note that Optim4RL surpasses VeLO (the state-of-the-art learned optimizer) significantly in Ant. This is a great success since VeLO is trained for 4,000 TPU-months on thousands of tasks while Optim4RL is only trained in six toy tasks for a few GPU-hours.

Table 5.3: The reward scales of gridworlds used for learning a general-purpose optimizer.

Gridworld	Reward Scale
small_dense_long	1000
small_dense_short	100
big_sparse_short	100
big_dense_short	10
big_sparse_long	10
big_dense_long	1

Finally, Optim4RL is also competitive compared with classical human-designed optimizers (Adam and RMSProp), even though it is entirely trained from scratch. Training a universally applicable learned optimizer for RL tasks is an inherently formidable challenge. Our results demonstrate the generalization ability of Optim4RL in complex unseen tasks, which is a great achievement in itself, proving the effectiveness of our approach.

5.5.3 Achieving Robust Training and Strong Generalization

Generally, we find it hard to train learned optimizers partly due to not-a-number (NaN) errors during training, even when gradient clipping or gradient normalization is applied. For example, among all meta-training hyper-parameter settings, we fail to train STAR due to NaN errors in more than 80% and 50% settings in Humanoid and Ant, respectively. However, NaN errors are seldom encountered when we meta-train Optim4RL, LinearOptim, and L2LGD^2 in Humanoid and Ant; and Optim4RL is the only one that achieves satisfactory performance among them.

Next, we show that Optim4RL not only generalizes to unseen tasks, but also transfers to different hyper-parameter settings. To be specific, we train our learned optimizer Optim4RL under the default hyper-parameter setting and then test it under different hyper-parameter settings in two gridworlds — small_dense_short and big_dense_long. We report the returns at the end of training, averaged over 10 runs. As shown in Table 5.4, Table 5.5, and Table 5.6, Optim4RL is robust under different hyper-parameter settings, such as GAE λ, entropy weight, and discount factor.

Table 5.4: The performance of Optim4RL with different GAE λ values in two gridworlds. All results are averaged over 10 runs, reported with 90% confidence intervals.

Task	Parameter Value	Return
small_dense_short	0.9	11.51±0.19
small_dense_short	0.95	11.25±0.16
small_dense_short	0.99	10.81±0.17
small_dense_short	0.995	10.66±0.17
big_dense_long	0.9	23.35±0.76
big_dense_long	0.95	23.57±0.60
big_dense_long	0.99	21.31±0.64
big_dense_long	0.995	20.55±0.54

Table 5.5: The performance of Optim4RL with different entropy weights in two gridworlds. All results are averaged over 10 runs, reported with 90% confidence intervals.

Task	Parameter Value	Return
small_dense_short	0.005	11.01±0.16
small_dense_short	0.01	11.13±0.09
small_dense_short	0.02	11.29±0.12
small_dense_short	0.04	11.25±0.16
big_dense_long	0.005	22.41±0.59
big_dense_long	0.01	22.45±0.79
big_dense_long	0.02	22.59±0.43
big_dense_long	0.04	19.96±1.30

Table 5.6: The performance of Optim4RL with different discount factors in two gridworlds. All results are averaged over 10 runs, reported with 90% confidence intervals.

Task	Parameter Value	Return
small_dense_short	0.9	12.47±0.14
small_dense_short	0.95	12.32±0.09
small_dense_short	0.99	11.48±0.09
small_dense_short	0.995	11.01±0.17
big_dense_long	0.9	18.13±3.27
big_dense_long	0.95	25.01±1.42
big_dense_long	0.99	25.45±0.47
big_dense_long	0.995	22.07±0.81

5.6 Conclusion

In this section, we analyzed the hardness of learning to optimize for RL and studied the failures of learned optimizers in RL. Our investigation reveals that agent-gradients in RL are non-IID and have high bias and variance. To mitigate these problems, we introduced pipeline training and a novel optimizer structure. Combining these techniques, we proposed a learned optimizer for RL, Optim4RL, which can be meta-learned to optimize RL tasks entirely from scratch. Although only trained in toy tasks, Optim4RL showed its strong generalization ability to unseen complex tasks.

Chapter 6

Model-free Policy Learning with Reward Gradients

Policy gradient methods are increasingly popular in the reinforcement learning (RL) community, with applications in computer games (Vinyals et al. 2019, Berner et al. 2019, Zha et al. 2021, Badia et al. 2020), simulated robotic tasks (Haarnoja et al. 2018, Lillicrap et al. 2016a), and recommendation systems (Zheng et al. 2018, Zhao et al. 2018, Afsar et al. 2022). However, they still suffer from low sample efficiency especially under non-stationarity, hindering their applicability to real-world situations. One way to mitigate this inefficiency, is to make best use of prior knowledge into the learning system, such as reward functions. Reward functions are usually known, allowing access to not only scalar reward signals but also reward gradients. Attempts to use reward gradients are already observed in prior works (Cai et al. 2020b, Heess et al. 2015, Hafner et al. 2019), but all these methods require a learned transition model which is hard to accurately obtain (van Hasselt et al. 2019, Jafferjee et al. 2020). In this chapter, we pose the following question: how can we leverage reward gradients to improve learning efficiency under non-stationarity without learning a model?

To answer this question, we develop a new policy gradient estimator—the *reward policy gradient* (RPG) estimator—that incorporates reward gradients. Specifically, our RPG estimator combines likelihood ratio (LR) and reparameterization (RP) gradients to avoid the need for a transition

model, while taking full advantage of reward gradients. This hybrid approach allows the RPG estimator to better track evolving reward landscapes and adapt policies accordingly. Based on this new estimator, we propose a new on-policy policy gradient algorithm—the RPG algorithm. We empirically show that by incorporating reward gradients into the gradient estimator, the bias and variance of estimated gradient decrease significantly, enabling more stable and efficient learning under non-stationarity. To analyze the benefit of reward gradients and the properties of our estimator, we test our algorithm on bandit and simple Markov decision processes, where the ground truth gradient is known in closed-form. Moreover, we compare RPG with a state-of-the-art actor-critic algorithm—proximal policy optimization (PPO)—on challenging problems, showing that our algorithm outperforms the baseline algorithm.

In the rest of this chapter, we first review the status of reward gradients prior to our work. We then move to our major theoretical result—the reward policy gradient theorem. Finally, we present RPG algorithm as well as the experimental results.

6.1 Only Model-Based Methods Use Reward Gradients So Far

In RL, policy gradient methods can be broadly classified into two categories: model-free and model-based algorithms. Model-free methods, such as REINFORCE (Williams 1992), TRPO (Schulman et al. 2015), and PPO (Schulman et al. 2017), estimate policy gradients directly from sampled trajectories without requiring an explicit transition model. These methods are generally more robust to model inaccuracies but suffer from high variance in gradient estimates and limited sample efficiency. In contrast, model-based methods requires a transition model to enable planning or improve sample efficiency.

Notably, several model-based methods have explored the use of reward gradients. For example, DVPG (Cai et al. 2020b) uses a learned deterministic model to backpropagate through the reward and value functions. SVG (Heess et al. 2015), which use a stochastic model of the dynamics and backpropagate gradients through sampled trajectories. Dreamer (Hafner et al. 2019), which learns a latent dynamics model to enable gradient-based optimization of policies through imagined rollouts.

Despite their innovations, all of these approaches rely on a learned transition model to compute gradients, which remains a challenging problem—particularly in high-dimensional or partially observable settings. Empirical studies have shown that model errors can accumulate during rollouts and destabilize training (van Hasselt et al. 2019, Jafferjee et al. 2020), sometimes outweighing the potential benefits of reward gradients. On the other hand, to the best of our knowledge, no existing model-free policy gradient algorithm has leveraged reward gradients without relying on a model.

6.2 Reward Policy Gradient Theorem

In this section, we first present our main theoretical result—the *reward policy gradient theorem*—a new policy gradient theorem that incorporates the gradient of the reward function without using a state transition function explicitly. The reward policy gradient theorem requires perfect knowledge of the reward and the value function. We show that an unbiased estimate can be obtained using approximated reward and state-value functions, by defining a set of compatible features, similarly to Sutton et al. (2000).

We begin by assuming that the action a is sampled from the policy π parameterized with θ given the current state s: $a \sim \pi_\theta(\cdot|s)$. We reparameterize the policy with a function f, $a = f_\theta(\epsilon; s)$, $\epsilon \sim p(\cdot)$. Let function g be the inverse function of f, that is, $\epsilon = g_\theta(a; s)$ and $a = f_\theta(g_\theta(a; s); s)$. Furthermore, following Imani et al. (2018), we make two common assumptions on the MDP.

We use the same objective $J(\theta)$ as in Equation (2.8). Then under these two assumptions, we have the following results.

Theorem 6.1 (Reward Policy Gradient)**.** *Suppose that the MDP satisfies Assumption 2.1, Assumption 2.2, and Assumption 2.3, then*

$$\nabla_\theta J(\theta) = \int d^{\pi_\theta}(s)\pi_\theta(a|s)\mathrm{P}(s'|s,a)\left[\nabla_\theta R(s, f_\theta(\epsilon; s))|_{\epsilon=g_\theta(a;s)} + \gamma V_{\pi_\theta}(s')\nabla_\theta \log \pi_\theta(a|s)\right] \mathrm{d}s\mathrm{d}a\mathrm{d}s',$$

where $d^{\pi_\theta}(s') = \int \sum_{t=0}^{\infty} \gamma^t p_0(s)\mathrm{P}(s \to s', t, \pi_\theta)\mathrm{d}s$ *is the (discounted) stationary state distribution for policy* π_θ *and* $\mathrm{P}(s \to s', t, \pi_\theta)$ *is the transition probability from* s *to* s' *with* t *steps under policy* π_θ*.*

Proof. We first apply the policy gradient theorem (Sutton et al. 2000) and get an intermediate result in form of the action-value function. Next, we split the action-value function in the policy gradient theorem into two parts: the immediate reward and the state-value of the next state. To incorporate reward gradients, we use the RP technique to the immediate reward part; to avoid the knowledge of the model, we apply the LR estimator to the state-value part. Finally, we combine both parts into an unbiased gradient estimation.

By the policy gradient theorem, we have

$$\nabla_\theta J(\theta) = \int d^{\pi_\theta}(s)\pi_\theta(a|s)Q_{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s.$$

Next, we split the action-value function $Q_{\pi_\theta}(s,a)$ into two parts:

$$\begin{aligned}
&\nabla_\theta J(\theta) \\
&= \int d^{\pi_\theta}(s)\pi_\theta(a|s)\left[R(s,a) + \gamma \int \mathrm{P}(s'|s,a)V_{\pi_\theta}(s')\mathrm{d}s'\right]\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}a\mathrm{d}s \\
&= \int d^{\pi_\theta}(s)\pi_\theta(a|s)R(s,a)\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a + \gamma \int d^{\pi_\theta}(s)\pi_\theta(a|s)\mathrm{P}(s'|s,a)V_{\pi_\theta}(s')\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a\mathrm{d}s' \\
&= \int d^{\pi_\theta}(s)R(s,a)\nabla_\theta \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a + \gamma \int d^{\pi_\theta}(s)\pi_\theta(a|s)\mathrm{P}(s'|s,a)V_{\pi_\theta}(s')\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a\mathrm{d}s' \\
&= \int d^{\pi_\theta}(s)\nabla_\theta \left(\int \pi_\theta(a|s)R(s,a)\mathrm{d}a\right)\mathrm{d}s + \gamma \int d^{\pi_\theta}(s)\pi_\theta(a|s)\mathrm{P}(s'|s,a)V_{\pi_\theta}(s')\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a\mathrm{d}s'.
\end{aligned}$$

Now, we apply the reparameterization technique to the first part,

$$\begin{aligned}
&\nabla_\theta J(\theta) \\
&= \int d^{\pi_\theta}(s)\nabla_\theta \left(\int \pi_\theta(a|s)R(s,a)\mathrm{d}a\right)\mathrm{d}s + \gamma \int d^{\pi_\theta}(s)\pi_\theta(a|s)\mathrm{P}(s'|s,a)V_{\pi_\theta}(s')\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a\mathrm{d}s' \\
&= \int d^{\pi_\theta}(s)\nabla_\theta \left(\int p(\epsilon)R(s,f_\theta(\epsilon;s))\mathrm{d}\epsilon\right)\mathrm{d}s + \gamma \int d^{\pi_\theta}(s)\pi_\theta(a|s)\mathrm{P}(s'|s,a)V_{\pi_\theta}(s')\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a\mathrm{d}s' \\
&= \int d^{\pi_\theta}(s)\left(\int p(\epsilon)\nabla_\theta R(s,f_\theta(\epsilon;s))\mathrm{d}\epsilon\right)\mathrm{d}s + \gamma \int d^{\pi_\theta}(s)\pi_\theta(a|s)\mathrm{P}(s'|s,a)V_{\pi_\theta}(s')\nabla_\theta \log \pi_\theta(a|s)\mathrm{d}s\mathrm{d}a\mathrm{d}s'.
\end{aligned}$$

We then apply the reverse operation of reparameterization to the first part,

$$
\begin{aligned}
&\nabla_\theta J(\theta) \\
&= \int d^{\pi_\theta}(s) \left(\int p(\epsilon) \nabla_\theta R(s, f_\theta(\epsilon; s)) \mathrm{d}\epsilon \right) \mathrm{d}s \\
&+ \gamma \int d^{\pi_\theta}(s) \pi_\theta(a|s) \mathrm{P}(s'|s,a) V_{\pi_\theta}(s') \nabla_\theta \log \pi_\theta(a|s) \mathrm{d}s\mathrm{d}a\mathrm{d}s' \\
&= \int d^{\pi_\theta}(s) \left(\int \pi_\theta(a|s) \nabla_\theta R(s, f_\theta(\epsilon; s))|_{\epsilon = g_\theta(a;s)} \mathrm{d}a \right) \mathrm{d}s \\
&+ \gamma \int d^{\pi_\theta}(s) \pi_\theta(a|s) \mathrm{P}(s'|s,a) V_{\pi_\theta}(s') \nabla_\theta \log \pi_\theta(a|s) \mathrm{d}s\mathrm{d}a\mathrm{d}s' \\
&= \int d^{\pi_\theta}(s) \pi_\theta(a|s) \mathrm{P}(s'|s,a) \left[\nabla_\theta R(s, f_\theta(\epsilon; s))|_{\epsilon = g_\theta(a;s)} + \gamma V_{\pi_\theta}(s') \nabla_\theta \log \pi_\theta(a|s) \right] \mathrm{d}s\mathrm{d}a\mathrm{d}s'.
\end{aligned}
$$

□

This theorem provides a new way of computing the objective gradients. Specifically, it presents the objective gradient in terms of both LR and RP gradients as additive components. The theorem also presents the first model-free unbiased gradient estimator of the objective function that utilizes gradients of the reward function. Some algorithms also estimate the gradient of the objective using gradients of the reward function, such as DVPG (Cai et al. 2020b) and SVG (Heess et al. 2015). However, they are model-based algorithms in the sense that they require the knowledge of the state transition function to estimate an unbiased gradient, while our theorem points out a model-free approach without the knowledge of the state transition.

Theorem 6.2 (Reward Policy Gradient with Function Approximation)**.** *Consider a parametric approximation of the reward function* $\hat{R}_\omega(s,a)$ *and a parametric approximation of the value function* $\hat{V}_\phi(s)$ *such that*

$$\nabla_a \hat{R}_\omega(s,a) = \nabla_\theta^\intercal f_\theta(\epsilon; s)|_{\epsilon = g_\theta(a;s)} \omega, \forall (s,a) \in \mathcal{S} \times \mathcal{A}$$

and

$$\int d^{\pi_\theta}(s) \pi_\theta(a|s) \mathrm{P}(s'|s,a) \nabla_\phi \hat{V}_\phi(s') \mathrm{d}s\mathrm{d}a = \int d^{\pi_\theta}(s) \pi_\theta(a|s) \mathrm{P}(s'|s,a) \nabla_\theta \log \pi_\theta(a|s) \mathrm{d}s\mathrm{d}a, \forall s' \in \mathcal{S}$$

where

$$\phi = \arg\min_{\phi} \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta\\ s'\sim \mathrm{P}(\cdot|s,a)}} \left[(\hat{V}_\phi(s') - V(s'))^2\right]$$

and

$$\omega = \arg\min_{\omega} \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[(\nabla_a \hat{R}_\omega(s,a) - \nabla_a R(s,a))^\intercal (\nabla_a \hat{R}_\omega(s,a) - \nabla_a R(s,a))\right].$$

Then,

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta\\ s'\sim \mathrm{P}(\cdot|s,a)}} \left[\nabla_\theta \hat{R}_\omega(s, f_\theta(\epsilon; s))|_{\epsilon=g_\theta(a;s)} + \gamma \hat{V}_\phi(s') \nabla_\theta \log \pi_\theta(a|s)\right].$$

Proof. By the definition of ω, we have

$$\begin{aligned}
&\omega = \arg\min_{\omega} \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[(\nabla_a \hat{R}_\omega(s,a) - \nabla_a R(s,a))^\intercal (\nabla_a \hat{R}_\omega(s,a) - \nabla_a R(s,a))\right] \\
&\implies \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[(\nabla_a \hat{R}_\omega(s,a) - \nabla_a R(s,a)) \nabla_\omega \nabla_a \hat{R}_\omega(s,a)\right] = 0 \\
&\implies \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[(\nabla_a \hat{R}_\omega(s,a) - \nabla_a R(s,a)) \nabla_\theta f_\theta(\epsilon; s)|_{\epsilon=g(a;s)}\right] = 0 \text{ (by the first assumption)} \\
&\implies \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[\nabla_a \hat{R}_\omega(s,a) \nabla_\theta f_\theta(\epsilon; s)|_{\epsilon=g(a;s)}\right] = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[\nabla_a R(s,a) \nabla_\theta f_\theta(\epsilon; s)|_{\epsilon=g(a;s)}\right] \\
&\implies \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[\nabla_\theta \hat{R}_\omega(s, f_\theta(\epsilon; s))|_{\epsilon=g_\theta(a;s)}\right] = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[\nabla_\theta R(s, f_\theta(\epsilon; s))|_{\epsilon=g_\theta(a;s)}\right].
\end{aligned}$$

By the second assumption, we have $\forall s' \in \mathcal{S}$,

$$\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[\mathrm{P}(s'|s,a) \nabla_\phi \hat{V}_\phi(s')\right] = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta}\\ a\sim\pi_\theta}} \left[\mathrm{P}(s'|s,a) \nabla_\theta \log \pi_\theta(a|s)\right].$$

Multiplying both sides by $\hat{V}_\phi(s') - V(s')$, we obtain the result for all $s' \in \mathcal{S}$,

$$
\begin{aligned}
&(\hat{V}_\phi(s') - V(s')) \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta}} \left[\mathrm{P}(s'|s,a)\nabla_\phi \hat{V}_\phi(s')\right] = (\hat{V}_\phi(s') - V(s')) \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta}} \left[\mathrm{P}(s'|s,a)\nabla_\theta \log \pi_\theta(a|s)\right] \\
\Longrightarrow &\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta}} \left[(\hat{V}_\phi(s') - V(s'))\mathrm{P}(s'|s,a)\nabla_\phi \hat{V}_\phi(s')\right] = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta}} \left[(\hat{V}_\phi(s') - V(s'))\mathrm{P}(s'|s,a)\nabla_\theta \log \pi_\theta(a|s)\right] \\
\Longrightarrow &\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[(\hat{V}_\phi(s') - V(s'))\nabla_\phi \hat{V}_\phi(s')\right] = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[(\hat{V}_\phi(s') - V(s'))\nabla_\theta \log \pi_\theta(a|s)\right].
\end{aligned}
$$

Note that the universal assumption ($\forall s' \in \mathcal{S}$) is required; without it, the derivations do not hold. Now considering the above equation and by the definitions of ϕ, we have

$$
\begin{aligned}
&\phi = \arg\min_\phi \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[(\hat{V}_\phi(s') - V(s'))^2\right] \\
\Longrightarrow &\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[(\hat{V}_\phi(s') - V(s'))\nabla_\phi \hat{V}_\phi(s')\right] = 0 \\
\Longrightarrow &\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[(\hat{V}_\phi(s') - V(s'))\nabla_\theta \log \pi_\theta(a|s)\right] = 0 \\
\Longrightarrow &\mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[\hat{V}_\phi(s')\nabla_\theta \log \pi_\theta(a|s)\right] = \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[V(s')\nabla_\theta \log \pi_\theta(a|s)\right].
\end{aligned}
$$

Combine the above results with Theorem 6.1, we have

$$
\begin{aligned}
&\nabla_\theta J(\theta) \\
&= \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[\nabla_\theta R(s, f_\theta(\epsilon; s))|_{\epsilon=g_\theta(a;s)} + \gamma V(s')\nabla_\theta \log \pi_\theta(a|s)\right] \\
&= \mathop{\mathbb{E}}_{\substack{s\sim d^{\pi_\theta} \\ a\sim\pi_\theta \\ s'\sim\mathrm{P}(\cdot|s,a)}} \left[\nabla_\theta \hat{R}_\omega(s, f_\theta(\epsilon; s))|_{\epsilon=g_\theta(a;s)} + \gamma \hat{V}_\phi(s')\nabla_\theta \log \pi_\theta(a|s)\right].
\end{aligned}
$$

□

By this theorem, we show that when the reward function and the state-value function are approximated by sufficiently good function approximators (e.g., neural networks), we can obtain an unbiased gradient estimation under certain assumptions. The functions $\hat{R}_\omega$ and $\hat{V}_\phi$ are also known as compatible approximators (Sutton et al. 2000, Peters and Schaal 2008). Note that ϕ, ω, and θ have same dimensions.

6.3 A Reward Policy Gradient Algorithm Based on PPO

Based on the reward policy gradient theorem, many different policy gradient algorithms can be developed that benefit from reward gradients without using a transition model. In this section, we develop a new policy gradient algorithm based on an existing deep policy gradient method called Proximal Policy Optimization (PPO). First, we introduce the baseline subtraction technique which is generally used in policy gradient algorithms to reduce variance. To be specific, we subtract the baseline $V_{\pi_\theta}(s)$ from $\gamma V_{\pi_\theta}(s')$ in the RPG estimator:

$$\nabla_\theta R(s, f_\theta(\epsilon; s)) + (\gamma V_{\pi_\theta}(s') - V_{\pi_\theta}(s)) \nabla_\theta \log \pi_\theta(a|s). \tag{6.1}$$

Note that no bias is introduced in this step (Sutton and Barto 2018).

In practice, when the state-value function and the reward function are unknown to the agent, we could use function approximators (e.g., neural networks) to approximate them. Furthermore, we use λ-return (Sutton and Barto 2018) G^λ_{t+1} to replace $V_{\pi_\theta}(s')$ in Equation (6.1), which has a close relationship to GAE $H_t^{\mathrm{GAE}(\lambda)}$ (Schulman et al. 2016), i.e., $G^\lambda_t = H_t^{\mathrm{GAE}(\lambda)} + V_{\pi_\theta}(s_t)$, where $\lambda \in [0, 1]$. Using λ-returns significantly reduce the variance of gradient estimations while retaining tolerable biases (Schulman et al. 2016).

A short introduction of PPO can be found in Section 2.2.2. The RPG algorithm builds on PPO with two major modifications. First, we replace the original LR estimator in PPO with the RPG estimator. Second, in order to use reward gradients, we have a neural network to learn the reward function. Basically, RPG can be viewed as a version of PPO but using the RPG estimator to do

gradient estimation. The detailed algorithm description for RPG is listed in Algorithm 6.

Algorithm 6 Reward Policy Gradient Algorithm (RPG)

Input: initial policy parameters θ, value estimate parameters ϕ, and reward estimate parameters w.
for $k = 1, 2, \ldots$ **do**
    Collect trajectories $\mathcal{D} = \{\tau_i\}$ with policy π_θ.

    Compute G_t and $G_t^\lambda = H_t^{\mathrm{GAE}(\lambda)} + \hat{V}_\phi(S_t)$.

    Compute PPO advantage $H_t = H_t^{\mathrm{GAE}(\lambda)}$ and normalize.
    for epoch $= 1, 2, \ldots$ **do**
        Slice trajectories $\mathcal{D}$ into mini-batches.
        for each mini-batch B **do**
            Set $\hat{\rho}_t(\theta)$ by Equation (2.13) and detach it from the computation graph.
            Reparameterize the action $A_t = f_\theta(\epsilon_t; S_t)$.
            Compute the predicted reward $\hat{r}_{t+1} = \hat{R}_w(S_t, f_\theta(\epsilon_t; S_t))$.

            Update θ by maximizing $\mathbb{E}_B[\hat{\rho}_t(\theta) H_t^{\mathrm{RPG}}]$, where $H_t^{\mathrm{RPG}} = \hat{r}_{t+1} + (\gamma G_{t+1}^\lambda - \hat{V}_\phi(S_t)) \times \log \pi_\theta(A_t|S_t)$.
            Update ϕ by minimizing $\mathbb{E}_B[(\hat{V}_\phi(S_t) - G_t)^2]$.
            Update w by minimizing $\mathbb{E}_B[(\hat{R}_w(S_t, A_t) - r_{t+1})^2]$.

6.4 Experiments

In this section, we first analyze the bias-variance trade-off for the RPG estimator on a linear quadratic Gaussian (LQG) control task. We then gain more understanding to the advantages and drawbacks of using reward gradients on two bandit tasks. Furthermore, we investigate the benefit of RPG when the reward function is known. Finally, we evaluate our algorithm on six MuJoCo control tasks (Todorov et al. 2012) and one robot task compared to the baseline method.

6.4.1 A Bias-Variance Analysis of the RPG Estimator

Environments can have highly stochastic rewards (Brockman et al. 2016). Knowing the reward function in advance allows reducing the stochasticity of the gradient estimator. Furthermore, in most cases, reparameterization gradients exhibit lower variance w.r.t. likelihood ratio gradients (Xu

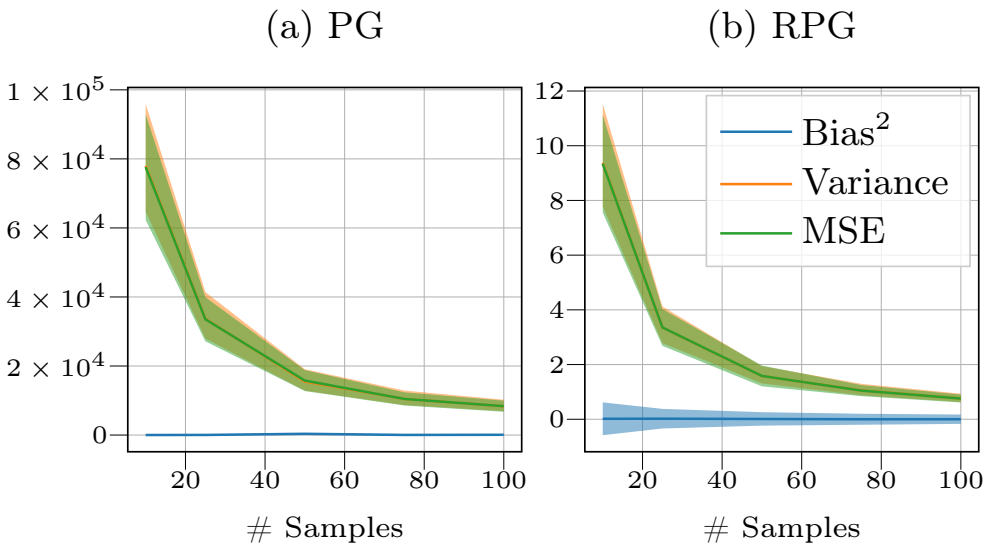


Figure 6.1: The bias, variance, and mean squared error (MSE) of the estimated gradient w.r.t. the number of samples for the PG estimator (left plot) and the RPG estimator (right plot). The shaded area representing a 95% interval using bootstrapping techniques. The values of bias, variance, and MSE for the RPG estimator are significantly smaller than the values for the PG estimator, which is clear from the Y-axis ranges.

et al. 2019). However, the variance of our estimator depends on many compounding factors (e.g. reward shapes and environment dynamics) that complicate a theoretical analysis. To compensate for this gap, we empirically investigate this aspect on a LQG problem.

The LQG control problem is one of the most classical control problems in control theory, with linear dynamics, quadratic reward, and Gaussian noise. Using the Riccati equations, we can solve the LQG problem in closed form. This property makes the LQG task an ideal benchmark to do bias-variance analysis for different policy gradient estimators. Specifically, we consider the discrete-time LQG problem, defined as

$$\max_{\theta} \sum_{t=0}^{\infty} \gamma^t r_t \quad \text{s.t. } \mathbf{s}_{t+1} = A\mathbf{s}_t + B\mathbf{a}_t; \quad r_t = -\mathbf{s}_t{}^\top Q\mathbf{s}_t - \mathbf{a}_t{}^\top Z\mathbf{a}_t$$

$$\mathbf{a}_{t+1} = \Theta\mathbf{s}_t + \Sigma\epsilon_t; \quad \epsilon_t \sim \mathcal{N}(0, I),$$

where A, B, Q, Z, Σ, and Θ are diagonal matrices and $\Theta = \text{diag}(\theta)$. Our policy is Gaussian: $\mathbf{a} \sim \mathcal{N}(\Theta\mathbf{s}; \Sigma)$, where $\theta \in \mathbb{R}^2$.

Given an LQG task, we study how the bias, the variance, and the mean squared error (MSE) of the estimated gradient vary w.r.t. the number of samples for both the policy gradient (PG) estimator and the RPG estimator. A similar study can also be found in Tosatto et al. (2021). Specifically, the discount factor is $\gamma = 0.99$. The number of steps for each episode is 100. The policy parameter is chosen randomly to be $\theta = [-1.1104430687690852, -1.3649958298432607]$. The parameters used for the LQG task are

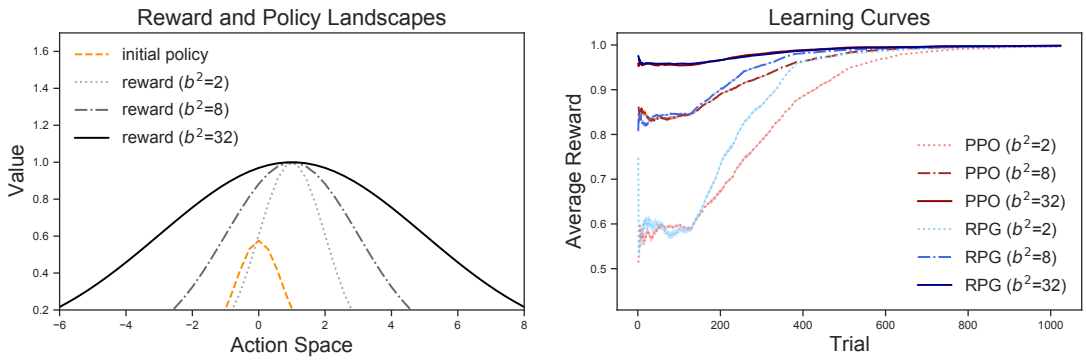
$$A = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}; \quad B = \begin{bmatrix} 1e-4 & 0 \\ 0 & 1e-4 \end{bmatrix}; \quad Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix};$$
$$R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}; \quad \mathbf{s}_0 = [0.5, 0.5].$$

In our experiment, we do not subtract a baseline term in either estimator as in Equation (6.1). Furthermore, we assume that the true reward function and the true value function are provided to both gradient estimators. Once the reward function is given, the RPG estimator is able to use not only the reward signals but also reward gradients. However, the PG estimator can only use the reward signals since it is not designed to use reward gradients.

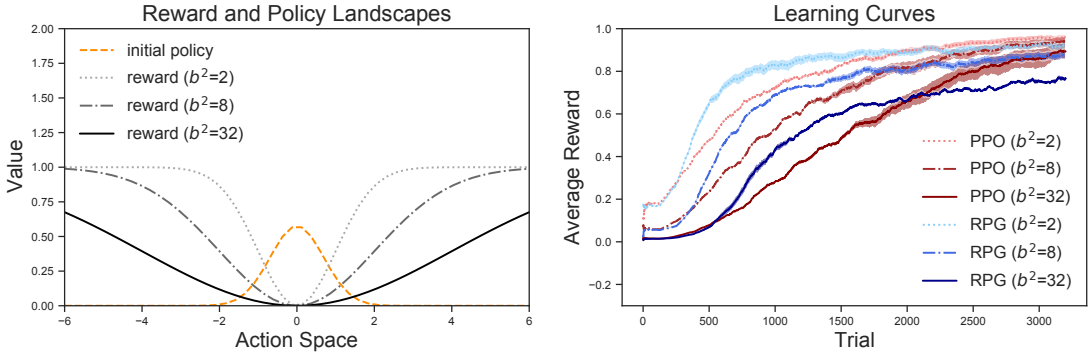
In Figure 6.1, the left and right plots show the values of the PG estimator and the RPG estimator, respectively. Clearly, both the bias and the variance of the RPG estimator are significantly smaller than the ones of the PG estimator, which can be noticed from the Y-axis ranges of the two plots. This result indicates that by incorporating reward gradients, the RPG estimator is able to reduce the bias and the variance of gradient estimation substantially.

6.4.2 Benefits and Drawbacks of Reward Gradients

To further understand the role of reward gradients, we design two kinds of bandit tasks—*Peaks* and *Holes*—with continuous action spaces . The reward functions of Peaks and Holes are defined as $r(a) = \exp\left(-\frac{(a-1)^2}{b^2}\right)$ and $r(a) = 1 - \exp\left(-\frac{a^2}{b^2}\right)$, as shown in Figure 6.2a and Figure 6.2b, respectively. In our experiments, a small ($0.01\times$) Gaussian noise is added to all reward signals and



(a) Peaks. RPG converges much faster than PPO in the whole training stage, when the variance is not so large.



(b) Holes. RPG converges much faster than PPO at the early training stage, under various variance settings. Then it slows down at the later stage.

Figure 6.2: The reward landscapes of Peaks and Holes as well as the learning curves for PPO and RPG during training. The initial Gaussian policies are also visualized. All results are averaged over 30 runs; the shaded areas represent standard errors. The plots show that the reward's gradient accelerates learning when it is large but hurts learning when the reward function is too flat.

b^2 is chosen from $[2, 8, 32]$.

The initial policy distribution is a Gaussian distribution $\mathcal{N}(0, 0.69)$. For RPG, we use a neural network to approximate the true reward function, denoted by $\hat{R}_w(a)$. In bandit cases, there are no value functions and we have the following single-sample gradients for PPO and RPG, without considering normalization for simplicity:

$$\begin{aligned}\nabla_\theta l_t^{\mathrm{PPO}}(\theta) &= \hat{\rho}_t(\theta) R_t \nabla_\theta \log \pi_\theta(A_t) \\ \nabla_\theta l_t^{\mathrm{RPG}}(\theta) &= \hat{\rho}_t(\theta) \nabla_\theta f_\theta(\epsilon_t) \nabla_a \hat{R}_w(a)|_{a=A_t},\end{aligned} \tag{6.2}$$

where $\theta = [\mu, \sigma]$. We test PPO and RPG on the bandits over 30 runs. The gradient is clipped by 1.

The learning curves of PPO and RPG are visualized in Figure 6.2a and Figure 6.2b. We observe that as b^2 increases, the learning speed decreases for both PPO and RPG, on Peaks and Holes. This is because the magnitude of reward's gradient decreases as b^2 increases which reduces $\nabla_\theta l_t(\theta)$ and slows down the learning update, as suggested by Equation (6.2).

Furthermore, RPG converges much faster than PPO on Peaks in the whole training period when b^2 is not so large. This shows that RPG is able to find the optimal action quicker than PPO, with the help of a relatively large reward's gradient. On Holes, RPG escapes from the sub-optimal action faster than PPO at the early training stage, but it slows down later and performs worse than PPO in the end. This is not surprising since the reward function is still sharp at the early training stage (when $|a|$ is small) but tends to become flat later (i.e., $|\nabla_a \hat{R}_w(a)|$ is smaller.) which slows down the learning process of RPG.

Therefore, we conclude that in bandits the reward's gradient accelerates learning when it is relatively large but hurts learning when the reward function is too flat. Reward and action-value gradients are identical in bandits; to compensate that, and we conduct the rest of the studies using simple and complex MDPs.

6.4.3 The Benefit of Knowing the Reward Function

To explore the role of reward gradients in MDP settings, we design a simple MDP with continuous state and action spaces—*Mountain Climbing*. Specifically, $\mathcal{S} = [-8, 8]^2$, $\mathcal{A} = [-1, 1]^2$, $s_0 = (0, 0), r_{t+1} = R(s_t, a_t)$, and $s_{t+1} = s_t + a_t + \epsilon$ where $R(s, a) = \exp(-||s + a - \nu||_2^2)$, $\epsilon \sim \mathcal{U}(-0.005, 0.005)$, and $\nu = (1, -1)$. Every episode ends in 10 steps. We test PPO (LR gradient) and RPG (LR gradient + reward gradient) on this MDP for $80,000$ steps. The number of epochs in PPO and RPG are to 1. The batch and mini-batch size are 40. The gradient is clipped by 0.5.

The average returns over 20 runs for each algorithm during training are shown in Figure 6.3. By utilizing reward gradients, RPG (LR gradient + reward gradient) outperforms PPO (LR gradient) significantly. Moreover, the reward gradients of the true reward function are more helpful to

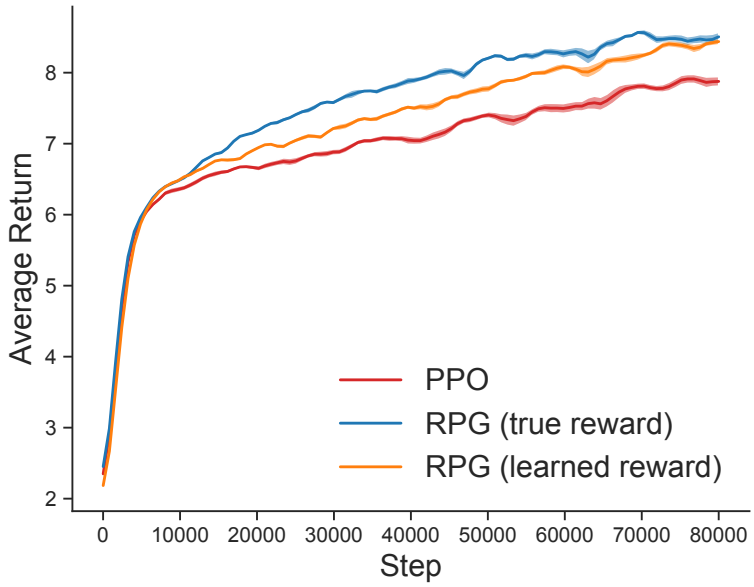


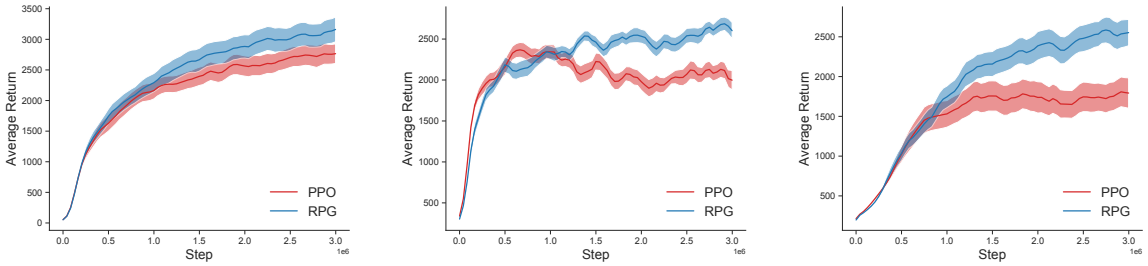
Figure 6.3: The learning curves for PPO and RPG during training on Mountain Climbing. The results are averaged over 20 runs, with the shaded area representing one standard error. In terms of convergence rate, RPG (true reward) $>$ RPG (learned reward) $>$ PPO.

accelerate learning than the reward gradients of a learned one, probably due to a higher accuracy of gradient estimation; although two versions of RPG reach a similar performance in the end.

6.4.4 Evaluation on MuJoCo Tasks

To further evaluate our algorithm, we measure its performance on six MuJoCo control tasks (Todorov 2014) through OpenAI Gym (Brockman et al. 2016). Our PPO implementation is based on Zhang (2018) and Achiam (2018), from which most hyper-parameters are adopted. Specifically, following the above two implementations, we use plain returns instead of λ-returns as the target values when computing the critic loss, as shown in Algorithm 2. We additionally tune certain PPO hyperparameters to align with the good performance reported in Zhang (2018) and Achiam (2018). Then we implement RPG based on this version of PPO. All hyper-parameters are listed in Table 6.1.

For the simulated tasks, each algorithm was trained on every task for 3 million steps. For every 5 epochs, we evaluated the agent's test performance using a deterministic policy for one episode.



(a) HalfCheetah-v2 (b) Hopper-v2 (c) Walker2d-v2

(d) Swimmer-v2 (e) Ant-v2 (f) Reacher-v2

Figure 6.4: The learning curves of evaluations on six benchmark tasks for PPO and RPG. The results are averaged over 30 runs, with the shaded area representing standard errors. RPG outperforms PPO significantly on three tasks – HalfCheetah, Hopper, and Walker2d.

Table 6.1: The hyper-parameter settings for PPO and RPG on MuJoCo tasks.

Hyper-parameter	**PPO**	**RPG**
Policy network learning rate	3×10^{-4}	3×10^{-4}
Value network learning rate	10^{-3}	10^{-3}
Reward network learning rate	None	10^{-3}
Hidden layers	[64, 64]	[64, 64]
Optimizer	Adam	Adam
Time-steps per iteration	2048	2048
Number of epochs	10	10
Mini-batch size	64	64
Discount factor (γ)	0.99	0.99
GAE parameter (λ)	0.95	0.95
PPO Clipping (ϵ)	0.2	0.2
Target KL divergence	0.01	0.01
State Clipping	[-10, 10]	[-10, 10]
Gradient clipping	2	2

Our results are reported by averaging over 30 runs with different random seeds. The learning curves during evaluation are presented in Figure 6.4. Overall, RPG outperforms PPO significantly on three tasks – HalfCheetah, Hopper, and Walker2d. It is slightly better on Reacher and worse on Ant, compared to PPO. On Swimmer, however, PPO has a clear advantage.

6.5 Discussion

Our work focuses more on understanding the properties of the RPG estimator by conducting a series of analysis on simple environments. Moreover, the RPG theorem only provides a new way to estimate policy gradient; the implementation of actual algorithms can vary. For example, the RPG version of the naïve actor-critic would be a straight-forward implementation. Our current implementation builds on PPO, and it benefits from PPO's techniques as well. To further explore and exploit the advantage of the RPG estimator, we may develop more advanced implementations by combining some modern techniques in the future, such as entropy regularization (Haarnoja et al. 2018), parallel training (Mnih et al. 2016), separating training phases for policy and value functions (Cobbe et al. 2021), Retrace (Munos et al. 2016), and V-trace (Espeholt et al. 2018), etc. Our current implementation is just one approach; exploring more possibilities are among the potential future works resulting from this work.

6.6 Conclusion

In this chapter, we introduced a novel strategy to compute the policy gradient which uses reward gradients without a model. Based on this strategy, we developed—RPG—a new on-policy policy gradient algorithm. We showed that our method of using reward gradients is beneficial over the PG estimator in terms of the bias-variance trade-off and sample efficiency. Experiments showed that RPG generally outperformed PPO on several simulation tasks.

Chapter 7

Conclusion

The final chapter begins by summarizing the key contributions of this thesis. We then discuss the limitations of our work and outline potential directions for future research. Finally, we conclude with a closing discussion.

7.1 Summary of Contributions

This thesis aimed to improve the learning efficiency of RL algorithms under non-stationarity, by proposing several approaches from diverse perspectives. The main contributions are presented across four core chapters, each tackling specific challenges in RL through novel algorithmic designs, architectural innovations, and theoretical insights.

- **MeDQN**: Two memory-efficient DQN variants that replace large replay buffers with state sampling, achieving high performance and sample efficiency with significantly lower memory usage.
- **Elephant**: A novel class of activation functions that mitigates catastrophic forgetting in neural networks, enhancing memory retention and learning performance of value-based RL algorithms.
- **Optim4RL**: A learned optimizer for RL that addresses the non-IID, high-bias, and high-

variance nature of agent-gradients in RL, enabling RL optimization from scratch and generalization to unseen tasks.

- **RPG**: A model-free policy gradient method that leverages reward gradients to improve the bias-variance trade-off and sample efficiency, outperforming PPO in continuous control tasks.

Collectively, these contributions offer a cohesive set of techniques aimed at enhancing the learning efficiency and robustness of RL under non-stationary conditions.

7.2 Limitations and Future Directions

The presented approaches in this thesis are still limited and could be improved potentially. In this section, we discuss the limitations of each contribution and explore future work for improvement.

MeDQN There are many open directions for future work. For example, a pre-trained encoder can be used to reduce high-dimensional inputs to low-dimensional inputs (Hayes et al. 2019; 2020, Chen et al. 2021), boosting the performance of MeDQN(U) on high-dimensional tasks. Combining classic memory-saving methods (Schlegel et al. 2017) with our knowledge consolidation approach may further reduce memory requirement and improve sample efficiency. Generative models (e.g., VAE (Kingma and Welling 2013) or GAN (Goodfellow et al. 2014)) could be applied to approximate d^π. The challenge for this approach would be to generate realistic pseudo-states to improve knowledge consolidation and reduce forgetting. It is also worth considering the usage of various sampling methods for the knowledge consolidation loss, such as stochastic gradient Langevin dynamics methods (Pan et al. 2022b; 2020). A combination of uniform state sampling and real state sampling might further improve the agent's performance. Finally, extending our ideas to policy gradient methods would also be interesting.

Elephant While our results demonstrate the effectiveness of elephant activation functions in model-free RL, we have not yet explored its applicability in model-based RL. Another limitation is the lack of a principled method for selecting the hyper-parameters of elephant activation functions.

The optimal values for these parameters appear to depend on both the input data and architectural factors such as the number of input and output features in each layer. In practice, careful tuning a is required to achieve a favorable stability-plasticity trade-off.

Optim4RL Learning to optimize for RL is a challenging problem. Due to memory and computation constraints, our current result is limited since we can only train Optim4RL in a small number of toy tasks. In the future, by leveraging more computation and memory, we expect to extend our approach to a larger scale and improve the performance of Optim4RL by training in more tasks with diverse RL agents. Moreover, theoretically analyzing the convergence of learned optimizers is also an interesting topic. We hope our analysis and proposed method can inspire and benefit future research, paving the way for better learned optimizers for RL.

RPG RPG is limited to the same class of tasks where reparameterization applies and, thus, not directly applicable to tasks with discrete actions. However, combined with the Gumbel-Softmax technique (Jang et al. 2017), it is possible to apply RPG on discrete control tasks as well.

7.3 Final Discussion

My long-term research goal is to develop an intelligent agent capable of continually and efficiently extracting, accumulating, and leveraging knowledge in the real world. I believe that the ability to continually learn is essential for a learning agent. On the one hand, intelligent systems that cannot learn continually will soon be outdated, losing the ability to adapt to the changing world. On the other hand, continual learning agents could potentially learn new tasks more efficiently by generalizing existing knowledge to unseen scenarios. As the capacity of learning models continues to increase, the associated training cost also rises. Training models completely from scratch is becoming increasingly unfeasible. Consequently, there is a growing imperative to develop efficient continual learning agents to reduce the high training cost.

There are mainly two challenging issues in continual learning—catastrophic forgetting and loss

of plasticity. Addressing them together poses an even greater challenge, known as the stability-plasticity dilemma. This thesis mainly focused on mitigating the forgetting issue. However, achieving zero forgetting under limited resources and infinite incoming information is very likely impossible. In fact, completely avoiding forgetting typically requires a continual learning algorithm to solve an NP-hard problem (Knoblauch et al. 2020). Thus, under the setting of limited resources, a more practical goal should be aiming to achieving mild forgetting (Property 2.3). However, even with a lower bar——using deep neural networks and gradient descent algorithms——it still appears impossible to achieve this goal in a continual learning setting. Some form of memory-augmented networks or external memory modules might be necessary. Ultimately, we would need a memory system capable of flexibly and efficiently supporting retrieving, adding, updating, deleting, and transferring information between agents.

Compared with catastrophic forgetting, loss of plasticity is relatively easier to solve, even under limited resources. Specifically, it refers to the phenomenon of a neural network becoming less capable of learning new information over time, especially after long-time training. While catastrophic forgetting leads to the loss of already learned old knowledge in neural networks, loss of plasticity prevents neural networks from learning new knowledge in the first place. In RL, as training proceeds, inactive neurons increase due to non-stationary targets (Sokar et al. 2023), which reduces neural network expressivity and limits learning performance. Loss of plasticity also makes agents prone to overfit to early experiences, resulting in a slow learning process and poor learning performance in subsequent training (Igl et al. 2021, Nikishin et al. 2022). One effective approach to maintaining network plasticity is injecting randomness into the parameters during training (Ash and Adams 2020, Igl et al. 2021, Dohare et al. 2021, Nikishin et al. 2022, D'Oro et al. 2023, Sokar et al. 2023, Dohare et al. 2024, Elsayed and Mahmood 2024). Adding regularization (Kumar et al. 2025, Lewandowski et al. 2023, Elsayed et al. 2024, Chung et al. 2024, Lewandowski et al. 2025) could also help maintaining plasticity significantly. However, most of these methods have only been evaluated on small-scale neural networks. Their effectiveness remains to be verified on large-scale architectures, such as large transformers and diffusion models.

Finally, in this thesis, we proposed a meta-learning approach to learn an optimizer for RL. A

more ambitious goal would be to meta-learn the entire learning process—including, but not limited to, the activation function, network architecture, loss function, optimizer, exploration strategy, and policy parameterization. With sufficient computational resources and data, this approach could potentially lead to the development of a strong continual learning agent.

References

Abbasi, A., Pirsiavash, H., Nooralinejad, P., Braverman, V., and Kolouri, S. (2022). Sparsity and heterogeneous dropout for continual learning in the null space of neural activations. In *Conference on Lifelong Learning Agents*. (p. 63.)

Achiam, J. (2018). Spinning up in deep reinforcement learning. https://github.com/openai/spinningup. (p. 113.)

Afsar, M. M., Crump, T., and Far, B. (2022). Reinforcement learning based recommender systems: A survey. *ACM Computing Surveys*. (p. 100.)

Aitchison, M., Sweetser, P., and Hutter, M. (2023). Atari-5: Distilling the arcade learning environment down to five games. In *International Conference on Machine Learning*. (pp. 51 and 70.)

Aljundi, R., Chakravarty, P., and Tuytelaars, T. (2017). Expert gate: Lifelong learning with a network of experts. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. (pp. 34 and 62.)

Aljundi, R., Lin, M., Goujaud, B., and Bengio, Y. (2019a). Gradient based sample selection for online continual learning. *Advances in neural information processing systems*. (p. 34.)

Aljundi, R., Rohrbach, M., and Tuytelaars, T. (2019b). Selfless sequential learning. In *International Conference on Learning Representations*. (p. 34.)

Alt, B., Šošić, A., and Koeppl, H. (2019). Correlation priors for reinforcement learning. *Advances in Neural Information Processing Systems*. (pp. 29 and 82.)

Ammar, H. B., Eaton, E., Ruvolo, P., and Taylor, M. (2014). Online multi-task learning for policy gradient methods. In *International Conference on Machine Learning*. (pp. 34 and 62.)

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., Shillingford, B., and De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. *Advances in Neural Information Processing Systems*. (pp. 27, 81, 85, 91, and 94.)

Ash, J. and Adams, R. P. (2020). On warm-starting neural network training. *Advances in neural information processing systems*. (p. 119.)

Atkinson, C., McCane, B., Szymanski, L., and Robins, A. (2021). Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting. *Neurocomputing*. (pp. 2, 24, and 35.)

Ba, J. and Caruana, R. (2014). Do deep nets really need to be deep? *Advances in Neural Information Processing Systems*. (p. 27.)

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. In *NIPS 2016 Deep Learning Symposium*. (p. 64.)

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., and Blundell, C. (2020). Agent57: Outperforming the atari human benchmark. In *International Conference on Machine Learning*. (pp. 1 and 100.)

Bagheri, S., Thill, M., Koch, P., and Konen, W. (2014). Online adaptable learning rates for the game connect-4. *IEEE Transactions on Computational Intelligence and AI in Games*. (p. 28.)

Banerjee, A., Cisneros-Velarde, P., Zhu, L., and Belkin, M. (2023). Neural tangent kernel at initialization: linear width suffices. In *Uncertainty in Artificial Intelligence*. (p. 25.)

Bechtle, S., Molchanov, A., Chebotar, Y., Grefenstette, E., Righetti, L., Sukhatme, G., and Meier, F. (2021). Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*. (pp. 27 and 28.)

Belfer, Y., Geifman, A., Galun, M., and Basri, R. (2024). Spectral analysis of the neural tangent kernel for deep residual networks. *Journal of Machine Learning Research*. (p. 25.)

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.* (pp. 51 and 70.)

Bengio, E., Pineau, J., and Precup, D. (2020a). Correcting momentum in temporal difference learning. *NeurIPS Workshop on Deep RL.* (pp. 82 and 85.)

Bengio, E., Pineau, J., and Precup, D. (2020b). Interference and generalization in temporal difference learning. In *International Conference on Machine Learning.* (p. 82.)

Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680.* (pp. 1 and 100.)

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. (2020). What is the state of neural network pruning? *Proceedings of machine learning and systems.* (p. 63.)

Boullé, N., Nakatsukasa, Y., and Townsend, A. (2020). Rational neural networks. *Advances in Neural Information Processing Systems.* (p. 91.)

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs. (pp. 69, 84, and 92.)

Bricken, T., Davies, X., Singh, D., Krotov, D., and Kreiman, G. (2023). Sparse distributed memory is a continual learner. In *International Conference on Learning Representations.* (p. 62.)

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. *arXiv preprint arXiv:1606.01540.* (pp. 45, 108, and 113.)

Buciluǎ, C., Caruana, R., and Niculescu-Mizil, A. (2006). Model compression. In *International Conference on Knowledge Discovery and Data Mining.* (p. 27.)

Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. (2020a). Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*. (p. 27.)

Cai, Q., Pan, L., and Tang, P. (2020b). Deterministic value-policy gradients. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (pp. 16, 100, 101, and 104.)

Carvalho, J., Tateo, D., Muratore, F., and Peters, J. (2021). An empirical analysis of measure-valued derivatives for policy gradients. In *2021 International Joint Conference on Neural Networks*. (p. 10.)

Ceron, J. S. O., Courville, A., and Castro, P. S. (2024). In value-based deep reinforcement learning, a pruned network is a good network. In *International Conference on Machine Learning*. (p. 63.)

Chaudhry, A., Rohrbach, M., Elhoseiny, M., Ajanthan, T., Dokania, P. K., Torr, P. H., and Ranzato, M. (2019). On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*. (p. 34.)

Chen, L., Lee, K., Srinivas, A., and Abbeel, P. (2021). Improving computational efficiency in visual reinforcement learning via stored embeddings. *Advances in Neural Information Processing Systems*. (p. 117.)

Chen, S., He, H., and Su, W. (2020). Label-aware neural tangent kernel: Toward better generalization and local elasticity. *Advances in Neural Information Processing Systems*. (pp. 25 and 63.)

Chen, Y., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Lillicrap, T. P., Botvinick, M., and de Freitas, N. (2017). Learning to learn without gradient descent by gradient descent. *International Conference on Machine Learning*. (p. 81.)

Chizat, L., Oyallon, E., and Bach, F. (2019). On lazy training in differentiable programming. *Advances in neural information processing systems*. (p. 66.)

Cho, K., van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural

machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation.* (p. 93.)

Choi, Y., El-Khamy, M., and Lee, J. (2021). Dual-teacher class-incremental learning with data-free generative replay. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* (p. 34.)

Chung, W., Cherif, L., Precup, D., and Meger, D. (2024). Parseval regularization for continual reinforcement learning. *Advances in Neural Information Processing Systems.* (p. 119.)

Cobbe, K. W., Hilton, J., Klimov, O., and Schulman, J. (2021). Phasic policy gradient. In *International Conference on Machine Learning.* (p. 115.)

Conti, E., Madhavan, V., Such, F. P., Lehman, J., Stanley, K. O., and Clune, J. (2018). Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Advances in Neural Information Processing Systems.* (p. 10.)

Cooper, C. (2000). On the rank of random matrices. *Random Structures & Algorithms.* (p. 38.)

Dabney, W., Barreto, A., Rowland, M., Dadashi, R., Quan, J., Bellemare, M. G., and Silver, D. (2021). The value-improvement path: Towards better representations for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence.* (p. 68.)

Daniel, C., Taylor, J., and Nowozin, S. (2016). Learning step size controllers for robust neural network training. In *Proceedings of the AAAI Conference on Artificial Intelligence.* (p. 28.)

Delange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., and Tuytelaars, T. (2021). A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* (pp. 2, 24, and 33.)

Deng, L. (2012). The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine.* (p. 88.)

Dettmers, T. and Zettlemoyer, L. (2019). Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840.* (p. 63.)

Doan, T., Bennani, M. A., Mazoure, B., Rabusseau, G., and Alquier, P. (2021). A theoretical analysis of catastrophic forgetting through the ntk overlap matrix. In *International Conference on Artificial Intelligence and Statistics*. (pp. 24 and 25.)

Dohare, S., Hernandez-Garcia, J. F., Lan, Q., Rahman, P., Mahmood, A. R., and Sutton, R. S. (2024). Loss of plasticity in deep continual learning. *Nature*. (pp. 1 and 119.)

Dohare, S., Sutton, R. S., and Mahmood, A. R. (2021). Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325*. (p. 119.)

D'Oro, P., Schwarzer, M., Nikishin, E., Bacon, P.-L., Bellemare, M. G., and Courville, A. (2023). Sample-efficient reinforcement learning by breaking the replay ratio barrier. In *International Conference on Learning Representations*. (p. 119.)

Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). RL^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*. (p. 28.)

Elsayed, M., Lan, Q., Lyle, C., and Mahmood, A. R. (2024). Weight clipping for deep continual and reinforcement learning. In *Reinforcement Learning Conference*. (p. 119.)

Elsayed, M. and Mahmood, A. R. (2024). Addressing loss of plasticity and catastrophic forgetting in continual learning. In *International Conference on Learning Representations*. (p. 119.)

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*. (p. 115.)

Farajtabar, M., Azizan, N., Mott, A., and Li, A. (2020). Orthogonal gradient descent for continual learning. In *International Conference on Artificial Intelligence and Statistics*. (pp. 34 and 54.)

Farquhar, S. and Gal, Y. (2018). Towards robust evaluations of continual learning. *arXiv preprint arXiv:1805.09733*. (pp. 2, 24, and 33.)

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., Pritzel, A., and Wierstra, D. (2017). PathNet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*. (pp. 34 and 62.)

Finn, C., Abbeel, P., and Levine, S. (2017a). Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*. (p. 27.)

Finn, C., Yu, T., Zhang, T., Abbeel, P., and Levine, S. (2017b). One-shot visual imitation learning via meta-learning. In *Conference on robot learning*. (p. 27.)

Fort, S., Nowak, P. K., Jastrzebski, S., and Narayanan, S. (2020). Stiffness: A new perspective on generalization in neural networks. *arXiv preprint arXiv:1901.09491*. (p. 74.)

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Hessel, M., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2018). Noisy networks for exploration. In *International Conference on Learning Representations*. (p. 71.)

Franceschi, L., Frasconi, P., Salzo, S., Grazzi, R., and Pontil, M. (2018). Bilevel programming for hyperparameter optimization and meta-learning. In *International Conference on Machine Learning*. (p. 27.)

Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*. (p. 63.)

Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., and Bachem, O. (2021). Brax - a differentiable physics engine for large scale rigid body simulation. (p. 92.)

French, R. M. (1992). Semi-distributed representations and catastrophic forgetting in connectionist networks. *Connection Science*. (p. 63.)

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*. (pp. 2 and 24.)

Gal, Y. (2016). *Uncertainty in deep learning*. PhD thesis, University of Cambridge. (p. 15.)

Ghiassian, S., Rafiee, B., Lo, Y. L., and White, A. (2020). Improving performance in reinforcement learning by breaking generalization in neural networks. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. (pp. 2, 29, and 66.)

Glynn, P. W. (1990). Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*. (p. 11.)

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*. (p. 117.)

Goodfellow, I., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. In *International Conference on Machine Learning*. (pp. 62, 63, and 69.)

Grathwohl, W., Choi, D., Wu, Y., Roeder, G., and Duvenaud, D. (2018). Backpropagation through the void: Optimizing control variates for black-box gradient estimation. In *International Conference on Learning Representations*. (p. 16.)

Greensmith, E., Bartlett, P. L., and Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*. (p. 14.)

Guo, Y., Yao, A., and Chen, Y. (2016). Dynamic network surgery for efficient DNNs. *Advances in neural information processing systems*. (p. 63.)

Gupta, A., Mendonca, R., Liu, Y., Abbeel, P., and Levine, S. (2018). Meta-reinforcement learning of structured exploration strategies. *Advances in Neural Information Processing Systems*. (p. 28.)

Ha, D. and Schmidhuber, J. (2018). Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*. (p. 29.)

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. (2018). Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*. (pp. 1, 16, 21, 29, 30, 100, and 115.)

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. (2019). Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*. (pp. 100 and 101.)

Han, S., Mao, H., and Dally, W. J. (2016). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*. (p. 27.)

Harrison, J., Metz, L., and Sohl-Dickstein, J. (2022). A closer look at learned optimization: Stability, robustness, and inductive biases. In *Advances in Neural Information Processing Systems*. (pp. 85, 88, 90, and 94.)

Hayes, T. L., Cahill, N. D., and Kanan, C. (2019). Memory efficient experience replay for streaming learning. In *International Conference on Robotics and Automation*. (pp. 1, 30, 34, and 117.)

Hayes, T. L., Kafle, K., Shrestha, R., Acharya, M., and Kanan, C. (2020). Remind your neural network to prevent catastrophic forgetting. In *European Conference on Computer Vision*. (p. 117.)

Hayes, T. L. and Kanan, C. (2022). Online continual learning for embedded devices. In *Conference on Lifelong Learning Agents*. (pp. 1 and 30.)

He, H. and Su, W. (2020). The local elasticity of neural networks. In *International Conference on Learning Representations*. (pp. 24 and 63.)

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Erez, T., and Tassa, Y. (2015). Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*. (pp. 16, 29, 100, 101, and 104.)

Henderson, P., Romoff, J., and Pineau, J. (2018). Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods. In *The 14th European Workshop on Reinforcement Learning*. (p. 84.)

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot,

B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (p. 69.)

Hinton, G., Vinyals, O., and Dean, J. (2014). Distilling the knowledge in a neural network. In *NIPS Deep Learning Workshop*. (pp. 26, 34, and 35.)

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*. (p. 85.)

Hoffman, M. D., Blei, D. M., Wang, C., and Paisley, J. (2013). Stochastic variational inference. *Journal of Machine Learning Research*. (p. 10.)

Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. (2021). Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*. (p. 27.)

Houthooft, R., Chen, Y., Isola, P., Stadie, B., Wolski, F., Jonathan Ho, O., and Abbeel, P. (2018). Evolved policy gradients. *Advances in Neural Information Processing Systems*. (pp. 27 and 28.)

Hsu, Y.-C., Liu, Y.-C., Ramasamy, A., and Kira, Z. (2018). Re-evaluating continual learning scenarios: A categorization and case for strong baselines. *arXiv preprint arXiv:1810.12488*. (pp. 2 and 24.)

Huang, J. and Yau, H.-T. (2020). Dynamics of deep neural networks and neural tangent hierarchy. In *International Conference on Machine Learning*. (p. 25.)

Huang, K., Wang, Y., Tao, M., and Zhao, T. (2020). Why do deep residual networks generalize better than deep feedforward networks?—a neural tangent kernel perspective. *Advances in neural information processing systems*. (p. 25.)

Huang, M., You, Y., Chen, Z., Qian, Y., and Yu, K. (2018). Knowledge distillation for sequence model. In *Interspeech*. (p. 27.)

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2018). Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*. (p. 27.)

Hung, C.-Y., Tu, C.-H., Wu, C.-E., Chen, C.-H., Chan, Y.-M., and Chen, C.-S. (2019). Compacting, picking and growing for unforgetting continual learning. *Advances in Neural Information Processing Systems*. (p. 62.)

Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, D., Tsounis, V., Koltun, V., and Hutter, M. (2019). Learning agile and dynamic motor skills for legged robots. *Science Robotics*. (p. 1.)

Igl, M., Farquhar, G., Luketina, J., Boehmer, W., and Whiteson, S. (2021). Transient non-stationarity and generalisation in deep reinforcement learning. In *International Conference on Learning Representations*. (pp. 1 and 119.)

Imani, E., Graves, E., and White, M. (2018). An off-policy policy gradient theorem using emphatic weightings. *Advances in Neural Information Processing Systems*. (pp. 11 and 102.)

Isele, D. and Cosgun, A. (2018). Selective experience replay for lifelong learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (p. 34.)

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural Networks*. (pp. 28 and 85.)

Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*. (pp. 25 and 66.)

Jafferjee, T., Imani, E., Talvitie, E., White, M., and Bowling, M. (2020). Hallucinating value: A pitfall of Dyna-style planning with imperfect environment models. *arXiv preprint arXiv:2006.04363*. (pp. 100 and 102.)

Jang, E., Gu, S., and Poole, B. (2017). Categorical reparameterization with Gumbel-Softmax. In *International Conference on Learning Representations*. (pp. 15 and 118.)

Javed, K. and White, M. (2019). Meta-learning representations for continual learning. *Advances in neural information processing systems*. (p. 27.)

Jin, X., Sadhu, A., Du, J., and Ren, X. (2020). Gradient based memory editing for task-free continual learning. In *4th Lifelong Machine Learning Workshop at ICML 2020*. (p. 34.)

Jung, D., Lee, D., Hong, S., Jang, H., Bae, H., and Yoon, S. (2023). New insights for the stability-plasticity dilemma in online continual learning. In *International Conference on Learning Representations*. (p. 64.)

Kamra, N., Gupta, U., and Liu, Y. (2017). Deep generative dual memory network for continual learning. *arXiv preprint arXiv:1710.10368*. (pp. 34 and 35.)

Kaplanis, C., Shanahan, M., and Clopath, C. (2019). Policy consolidation for continual reinforcement learning. In *International Conference on Machine Learning*. (p. 35.)

Kemker, R., McClure, M., Abitino, A., Hayes, T., and Kanan, C. (2018). Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (p. 33.)

Khetarpal, K., Riemer, M., Rish, I., and Precup, D. (2022). Towards continual reinforcement learning: A review and perspectives. *Journal of Artificial Intelligence Research*. (pp. 2, 24, 33, 34, and 87.)

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations*. (pp. 65, 71, 86, 90, and 91.)

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*. (pp. 10, 16, and 117.)

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*. (pp. 34 and 35.)

Kirsch, L., Flennerhag, S., van Hasselt, H., Friesen, A., Oh, J., and Chen, Y. (2022). Introducing symmetries to black box meta reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (p. 28.)

Kirsch, L., van Steenkiste, S., and Schmidhuber, J. (2020). Improving generalization in meta

reinforcement learning using learned objectives. In *International Conference on Learning Representations*. (pp. 27 and 28.)

Knoblauch, J., Husain, H., and Diethe, T. (2020). Optimal continual learning has perfect memory and is np-hard. In *International Conference on Machine Learning*. (p. 119.)

Kumar, S., Marklund, H., and Van Roy, B. (2025). Maintaining plasticity in continual learning via regenerative regularization. In *Conference on Lifelong Learning Agents*. (p. 119.)

Lai, K.-H., Zha, D., Li, Y., and Hu, X. (2020). Dual policy distillation. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. (p. 27.)

Lan, Q. (2019). A pytorch reinforcement learning framework for exploring new ideas. https://github.com/qlan3/Explorer. (p. 69.)

Lan, Q. and Mahmood, A. R. (2023). Elephant neural networks: Born to be a continual learner. *ICML Workshop on High-dimensional Learning Dynamics*. (p. iii.)

Lan, Q., Mahmood, A. R., YAN, S., and Xu, Z. (2024). Learning to optimize for reinforcement learning. *Reinforcement Learning Journal*. (p. iii.)

Lan, Q., Pan, Y., Fyshe, A., and White, M. (2020). Maxmin Q-learning: Controlling the estimation bias of q-learning. In *International Conference on Learning Representations*. (p. 45.)

Lan, Q., Pan, Y., Luo, J., and Mahmood, A. R. (2023). Memory-efficient reinforcement learning with value-based knowledge consolidation. *Transactions on Machine Learning Research*. (pp. iii and 1.)

Lan, Q., Tosatto, S., Farrahi, H., and Mahmood, R. (2022). Model-free policy learning with reward gradients. In *International Conference on Artificial Intelligence and Statistics*. (p. iii.)

Le, L., Kumaraswamy, R., and White, M. (2017). Learning sparse representations in reinforcement learning with sparse coding. *International Joint Conferences on Artificial Intelligence*. (p. 63.)

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*. (pp. 21 and 81.)

L'ecuyer, P. (1990). A unified view of the ipa, sf, and lr gradient estimation techniques. *Management Science*. (p. 10.)

Lee, J., Kim, S., Kim, S., Jo, W., and Yoo, H.-J. (2021). GST: Group-sparse training for accelerating deep reinforcement learning. *arXiv preprint arXiv:2101.09650*. (p. 63.)

Lewandowski, A., , Kumar, S., Schuurmans, D., György, A., and Machado, M. C. (2025). Learning continually by spectral regularization. In *International Conference on Learning Representations*. (p. 119.)

Lewandowski, A., Tanaka, H., Schuurmans, D., and Machado, M. C. (2023). Directions of curvature as an explanation for loss of plasticity. *arXiv preprint arXiv:2312.00246*. (p. 119.)

Li, A. and Pathak, D. (2021). Functional regularization for reinforcement learning via learned Fourier features. *Advances in Neural Information Processing Systems*. (p. 64.)

Li, K. and Malik, J. (2017). Learning to optimize. In *International Conference on Learning Representations*. (pp. 28 and 85.)

Li, X., Zhou, Y., Wu, T., Socher, R., and Xiong, C. (2019). Learn to grow: A continual structure learning framework for overcoming catastrophic forgetting. In *International Conference on Machine Learning*. (pp. 34 and 62.)

Li, Z., Zhou, F., Chen, F., and Li, H. (2017). Meta-SGD: Learning to learn quickly for few-shot learning. *arXiv preprint arXiv:1707.09835*. (p. 27.)

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016a). Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*. (pp. 1, 16, and 100.)

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016b). Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*. (p. 29.)

Lin, G., Chu, H., and Lai, H. (2022). Towards better plasticity-stability trade-off in incremental learning: A simple linear connector. In *Conference on Computer Vision and Pattern Recognition*. (p. 64.)

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*. (pp. 22 and 29.)

Liu, H., Simonyan, K., and Yang, Y. (2019a). Darts: Differentiable architecture search. In *International Conference on Learning Representations*. (p. 27.)

Liu, J., Xu, Z., Shi, R., Cheung, R. C. C., and So, H. K. (2020). Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers. In *International Conference on Learning Representations*. (p. 63.)

Liu, V., Kumaraswamy, R., Le, L., and White, M. (2019b). The utility of sparse representations for control in reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (pp. 56, 63, 65, and 66.)

Lopez-Paz, D. and Ranzato, M. (2017). Gradient episodic memory for continual learning. *Advances in neural information processing systems*. (p. 34.)

Lu, C., Kuba, J., Letcher, A., Metz, L., Schroeder de Witt, C., and Foerster, J. (2022). Discovered policy optimisation. *Advances in Neural Information Processing Systems*. (pp. 28 and 93.)

Lu, L., Jin, P., Pang, G., Zhang, Z., and Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*. (p. 91.)

Lucas, J., Sun, S., Zemel, R., and Grosse, R. (2019). Aggregated momentum: Stability through passive damping. In *International Conference on Learning Representations*. (pp. 86 and 90.)

Lyle, C., Rowland, M., and Dabney, W. (2021). Understanding and preventing capacity loss in reinforcement learning. In *International Conference on Learning Representations*. (pp. 1 and 88.)

Lyle, C., Zheng, Z., Nikishin, E., Pires, B. A., Pascanu, R., and Dabney, W. (2023). Understanding plasticity in neural networks. In *International Conference on Machine Learning*. (pp. 64 and 74.)

Ma, X., Zhu, J., Lin, Z., Chen, S., and Qin, Y. (2022). A state-of-the-art survey on solving non-iid data in federated learning. *Future Generation Computer Systems*. (p. 87.)

Madireddy, S., Yanguas-Gil, A., and Balaprakash, P. (2023). Improving performance in continual learning tasks using bio-inspired architectures. In *Conference on Lifelong Learning Agents*. (p. 62.)

Maheswaranathan, N., Sussillo, D., Metz, L., Sun, R., and Sohl-Dickstein, J. (2021). Reverse engineering learned optimizers reveals known and novel mechanisms. *Advances in Neural Information Processing Systems*. (p. 81.)

Mahmood, A. R., Sutton, R. S., Degris, T., and Pilarski, P. M. (2012). Tuning-free step-size adaptation. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. (p. 85.)

Mallya, A. and Lazebnik, S. (2018). PackNet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. (pp. 34 and 62.)

Masana, M., Tuytelaars, T., and Van de Weijer, J. (2021). Ternary feature masks: zero-forgetting for task-incremental learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. (pp. 34 and 62.)

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*. (pp. 2 and 24.)

Medsker, L. R. and Jain, L. (2001). Recurrent neural networks. *Design and Applications*. (p. 85.)

Mehta, H., Cutkosky, A., and Neyshabur, B. (2021). Extreme memorization via scale of initialization. In *International Conference on Learning Representations*. (p. 63.)

Mendez, J., Wang, B., and Eaton, E. (2020). Lifelong policy gradient learning of factored policies for faster training without forgetting. *Advances in Neural Information Processing Systems*. (pp. 34, 54, and 62.)

Mendez, J. A., van Seijen, H., and Eaton, E. (2022). Modular lifelong reinforcement learning via neural composition. In *International Conference on Learning Representations*. (pp. 34 and 54.)

Mermillod, M., Bugaiska, A., and BONIN, P. (2013). The stability-plasticity dilemma: investigating the continuum from catastrophic forgetting to age-limited learning effects. *Frontiers in Psychology*. (p. 64.)

Metz, L., Freeman, C. D., Harrison, J., Maheswaranathan, N., and Sohl-Dickstein, J. (2022a). Practical tradeoffs between memory, compute, and performance in learned optimizers. In *Conference on Lifelong Learning Agents*. (p. 85.)

Metz, L., Harrison, J., Freeman, C. D., Merchant, A., Beyer, L., Bradbury, J., Agrawal, N., Poole, B., Mordatch, I., Roberts, A., et al. (2022b). VeLO: Training versatile learned optimizers by scaling up. *arXiv preprint arXiv:2211.09760*. (pp. 82, 86, 87, 94, and 96.)

Metz, L., Maheswaranathan, N., Freeman, C. D., Poole, B., and Sohl-Dickstein, J. (2020a). Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *arXiv preprint arXiv:2009.11243*. (pp. 86, 87, and 88.)

Metz, L., Maheswaranathan, N., Nixon, J., Freeman, D., and Sohl-Dickstein, J. (2019). Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*. (pp. 85 and 87.)

Metz, L., Maheswaranathan, N., Sun, R., Freeman, C. D., Poole, B., and Sohl-Dickstein, J. (2020b). Using a thousand optimization tasks to learn hyperparameter search strategies. *arXiv preprint arXiv:2002.11887*. (p. 82.)

Mirzadeh, S. I., Chaudhry, A., Yin, D., Hu, H., Pascanu, R., Gorur, D., and Farajtabar, M. (2022a). Wide neural networks forget less catastrophically. In *International Conference on Machine Learning*. (pp. 54 and 62.)

Mirzadeh, S. I., Chaudhry, A., Yin, D., Nguyen, T., Pascanu, R., Gorur, D., and Farajtabar, M. (2022b). Architecture matters in continual learning. *arXiv preprint arXiv:2202.00275*. (pp. 54 and 62.)

Mirzadeh, S. I., Farajtabar, M., and Ghasemzadeh, H. (2020). Dropout as an implicit gating mechanism for continual learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. (p. 63.)

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*. (pp. 15, 87, 92, and 115.)

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., and Antonoglou, I. (2013). Playing Atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. (pp. 29, 55, and 69.)

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*. (pp. 1, 21, 29, 30, and 69.)

Mohamed, S., Rosca, M., Figurnov, M., and Mnih, A. (2020). Monte Carlo gradient estimation in machine learning. *Journal of Machine Learning Research*. (pp. 10, 15, and 16.)

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. G. (2016). Safe and efficient off-policy reinforcement learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. (p. 115.)

Nagabandi, A., Clavera, I., Liu, S., Fearing, R. S., Abbeel, P., Levine, S., and Finn, C. (2019). Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. In *International Conference on Learning Representations*. (p. 28.)

Nikishin, E., Schwarzer, M., D'Oro, P., Bacon, P.-L., and Courville, A. (2022). The primacy bias in deep reinforcement learning. In *International Conference on Machine Learning*. (p. 119.)

Oh, J., Hessel, M., Czarnecki, W. M., Xu, Z., van Hasselt, H. P., Singh, S., and Silver, D. (2020). Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*. (pp. 27, 28, and 92.)

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepesvari, C., Singh, S., et al. (2020). Behaviour suite for reinforcement learning. In *International Conference on Learning Representations*. (p. 92.)

Ostapenko, O., Puscas, M., Klein, T., Jahnichen, P., and Nabi, M. (2019). Learning to remember: A synaptic plasticity driven framework for continual learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. (p. 62.)

Pan, Y., Banman, K., and White, M. (2022a). Fuzzy tiling activations: A simple approach to learning sparse representations online. In *International Conference on Learning Representations*. (pp. 2, 29, 62, 69, 70, and 71.)

Pan, Y., Mei, J., and massoud Farahmand, A. (2020). Frequency-based search-control in dyna. In *International Conference on Learning Representations*. (p. 117.)

Pan, Y., Mei, J., massoud Farahmand, A., White, M., Yao, H., Rohani, M., and Luo, J. (2022b). Understanding and mitigating the limitations of prioritized experience replay. In *Conference on Uncertainty in Artificial Intelligence*. (p. 117.)

Parmas, P., Rasmussen, C. E., Peters, J., and Doya, K. (2018). PIPPS: Flexible model-based policy search robust to the curse of chaos. In *International Conference on Machine Learning*. (p. 15.)

Peters, J. and Schaal, S. (2008). Reinforcement learning of motor skills with policy gradients. *Neural networks*. (p. 107.)

Pflug, G. C. (1989). Sampling derivatives of probabilities. *Computing*. (p. 10.)

Ramapuram, J., Gregorova, M., and Kalousis, A. (2020). Lifelong generative modeling. *Neurocomputing*. (p. 34.)

Rebuffi, S.-A., Kolesnikov, A., Sperl, G., and Lampert, C. H. (2017). iCaRL: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition.* (p. 33.)

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of the 31st International Conference on International Conference on Machine Learning.* (p. 16.)

Riemer, M., Cases, I., Ajemian, R., Liu, M., Rish, I., Tu, Y., and Tesauro, G. (2018). Learning to learn without forgetting by maximizing transfer and minimizing interference. In *International Conference on Learning Representations.* (pp. 33, 34, and 54.)

Riemer, M., Cases, I., Ajemian, R., Liu, M., Rish, I., Tu, Y., and Tesauro, G. (2019). Learning to learn without forgetting by maximizing transfer and minimizing interference. In *International Conference on Learning Representations.* (p. 25.)

Ring, M. B. (1994). *Continual learning in reinforcement environments.* The University of Texas at Austin. (p. 1.)

Rolnick, D., Ahuja, A., Schwarz, J., Lillicrap, T., and Wayne, G. (2019). Experience replay for continual learning. *Advances in Neural Information Processing Systems.* (p. 35.)

Rusu, A. A., Colmenarejo, S. G., Gulcehre, C., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. (2015). Policy distillation. *arXiv preprint arXiv:1511.06295.* (p. 27.)

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. (2016). Progressive neural networks. *arXiv preprint arXiv:1606.04671.* (pp. 34 and 62.)

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *5th Workshop on Energy Efficient Machine Learning and Cognitive Computing at NeurIPS 2019.* (p. 27.)

Sarfraz, F., Arani, E., and Zonooz, B. (2023). Sparse coding in a dual memory system for lifelong learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (p. 63.)

Sarigül, M. and Avci, M. (2018). Performance comparison of different momentum techniques on deep reinforcement learning. *Journal of Information and Telecommunication*. (p. 84.)

Schlegel, M., Pan, Y., Chen, J., and White, M. (2017). Adapting kernel representations online using submodular maximization. In *International Conference on Machine Learning*. (p. 117.)

Schmidhuber, J. (1987). *Evolutionary principles in self-referential learning*. PhD thesis, Technical University of Munich. (pp. 27 and 28.)

Schraudolph, N. and Sejnowski, T. J. (1995). Tempering backpropagation networks: Not all weights are created equal. *Advances in Neural Information Processing Systems*. (p. 28.)

Schraudolph, N. N. (1998). Online local gain adaptation for multi-layer perceptrons. *Technical Report, IDSIA-09–98, IDSIA*. (p. 28.)

Schraudolph, N. N. (1999). Local gain adaptation in stochastic gradient descent. *International Conference on Artificial Neural Networks*. (p. 28.)

Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*. (p. 28.)

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*. (pp. 1 and 29.)

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*. (pp. 15, 29, and 101.)

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*. (pp. 2, 14, 22, 84, and 107.)

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*. (pp. 15, 22, 29, 92, and 101.)

Schwarz, J., Czarnecki, W., Luketina, J., Grabska-Barwinska, A., Teh, Y. W., Pascanu, R., and Hadsell, R. (2018). Progress & compress: A scalable framework for continual learning. In *International Conference on Machine Learning*. (pp. 2, 24, and 34.)

Serra, J., Suris, D., Miron, M., and Karatzoglou, A. (2018). Overcoming catastrophic forgetting with hard attention to the task. In *International Conference on Machine Learning*. (pp. 34 and 62.)

Shazeer, N. and Stern, M. (2018). Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*. (p. 91.)

Shen, Y., Dasgupta, S., and Navlakha, S. (2021). Algorithmic insights on continual learning from fruit flies. *arXiv preprint arXiv:2107.07617*. (pp. 56 and 62.)

Shin, H., Lee, J. K., Kim, J., and Kim, J. (2017). Continual learning with deep generative replay. *Advances in neural information processing systems*. (pp. 34 and 35.)

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International Conference on Machine Learning*. (pp. 16 and 17.)

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine learning*. (p. 14.)

Smith, L., Kostrikov, I., and Levine, S. (2023). Demonstrating a walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *Robotics: Science and Systems (RSS) Demo*. (p. 30.)

Sokar, G., Agarwal, R., Castro, P. S., and Evci, U. (2023). The dormant neuron phenomenon in deep reinforcement learning. In *International Conference on Machine Learning*. (p. 119.)

Sokar, G., Mocanu, D. C., and Pechenizkiy, M. (2021). SpaceNet: Make free space for continual learning. *Neurocomputing*. (pp. 34, 62, and 63.)

Sokar, G., Mocanu, E., Mocanu, D. C., Pechenizkiy, M., and Stone, P. (2022). Dynamic sparse training for deep reinforcement learning. In *International Joint Conference on Artificial Intelligence*. (p. 63.)

Song, X., Gao, W., Yang, Y., Choromanski, K., Pacchiano, A., and Tang, Y. (2020). ES-MAML: Simple hessian-free meta learning. In *International Conference on Learning Representations*. (p. 28.)

Srivastava, R. K., Masci, J., Kazerounian, S., Gomez, F., and Schmidhuber, J. (2013). Compete to compute. *Advances in neural information processing systems*. (pp. 63 and 69.)

Sun, Y. and Fazli, P. (2019). Real-time policy distillation in deep reinforcement learning. *arXiv preprint arXiv:1912.12630*. (p. 27.)

Sutton, R. (1992a). Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (pp. 27, 28, 81, and 85.)

Sutton, R. S. (1992b). Gain adaptation beats least squares. In *Proceedings of the 7th Yale workshop on adaptive and learning systems*. (p. 28.)

Sutton, R. S. (2022). A history of meta-gradient: Gradient methods for meta-learning. *arXiv preprint arXiv:2202.09701*. (p. 28.)

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, second edition. (pp. 6, 11, 22, 32, 43, 63, and 107.)

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*. (pp. 102, 103, and 107.)

Tan, Y., Hu, P., Pan, L., Huang, J., and Huang, L. (2023). RLx2: Training a sparse deep reinforcement learning model from scratch. In *International Conference on Learning Representations*. (p. 63.)

Tang, R., Lu, Y., Liu, L., Mou, L., Vechtomova, O., and Lin, J. (2019). Distilling task-specific knowledge from BERT into simple neural networks. *arXiv preprint arXiv:1903.12136*. (p. 27.)

Tasfi, N. (2016). PyGame learning environment. https://github.com/ntasfi/PyGame-Learning-Environment. (pp. 45 and 69.)

Telgarsky, M. (2017). Neural networks and rational functions. In *International Conference on Machine Learning*. (p. 91.)

Thrun, S. and Pratt, L. (1998). Learning to learn: Introduction and overview. In *Learning to learn*. (p. 27.)

Tian, Y., Krishnan, D., and Isola, P. (2020). Contrastive representation distillation. In *International Conference on Learning Representations*. (p. 27.)

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA Neural Networks Neural Networks for Machine Learning*. (pp. 70, 83, 87, and 91.)

Titsias, M. K., Schwarz, J., Matthews, A. G. d. G., Pascanu, R., and Teh, Y. W. (2020). Functional regularisation for continual learning with gaussian processes. In *International Conference on Learning Representations*. (p. 35.)

Todorov, E. (2014). Convex and analytically-invertible dynamics with contacts and constraints: Theory and implementation in MuJoCo. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*. (p. 113.)

Todorov, E., Erez, T., and Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. (pp. 92 and 108.)

Tosatto, S., Carvalho, J., and Peters, J. (2021). Batch reinforcement learning with a nonparametric off-policy policy gradient. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. (p. 110.)

Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., Cola, G. d., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Shen, A. T. J., and Younis, O. G. (2023). Gymnasium. https://zenodo.org/record/8127025. (pp. 55 and 69.)

Van de Ven, G. M. and Tolias, A. S. (2018). Generative replay with feedback connections as a general strategy for continual learning. *arXiv preprint arXiv:1809.10635*. (p. 34.)

Van de Ven, G. M., Tuytelaars, T., and Tolias, A. S. (2022). Three types of incremental learning. *Nature Machine Intelligence*. (pp. 2, 24, and 33.)

Van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., and Modayil, J. (2018). Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*. (p. 88.)

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*. (p. 29.)

van Hasselt, H. P., Hessel, M., and Aslanides, J. (2019). When to use parametric models in reinforcement learning? *Advances in Neural Information Processing Systems*. (pp. 100 and 102.)

Vettoruzzo, A., Bouguelia, M.-R., Vanschoren, J., Rögnvaldsson, T., and Santosh, K. (2024). Advances and challenges in meta-learning: A technical review. *IEEE transactions on pattern analysis and machine intelligence*. (p. 27.)

Vicol, P., Metz, L., and Sohl-Dickstein, J. (2021). Unbiased gradient estimation in unrolled computation graphs with persistent evolution strategies. In *International Conference on Machine Learning*. (p. 85.)

Vilalta, R. and Drissi, Y. (2002). A perspective view and survey of meta-learning. *Artificial Intelligence Review*. (p. 27.)

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*. (pp. 1 and 100.)

Wang, H., Zheng, S., Xiong, C., and Socher, R. (2019). On the generalization gap in reparameterizable reinforcement learning. In *International Conference on Machine Learning*. (p. 16.)

Wang, L., Zhang, X., Su, H., and Zhu, J. (2024). A comprehensive survey of continual learning: Theory, method and application. *IEEE transactions on pattern analysis and machine intelligence*. (pp. 2 and 87.)

Wang, Y., Vasan, G., and Mahmood, A. R. (2023). Real-time reinforcement learning for vision-based robotics utilizing local and remote computers. In *International Conference on Robotics and Automation*. (p. 30.)

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 *International Conference on Learning Representations*. (p. 15.)

Wang, Z., Zhan, Z., Gong, Y., Yuan, G., Niu, W., Jian, T., Ren, B., Ioannidis, S., Wang, Y., and Dy, J. (2022). SparCL: Sparse continual learning on the edge. In *Advances in Neural Information Processing Systems*. (p. 63.)

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge. (p. 9.)

Weng, J., Chen, H., Yan, D., You, K., Duburcq, A., Zhang, M., Su, Y., Su, H., and Zhu, J. (2022). Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*. (pp. 51 and 70.)

Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Freitas, N., and Sohl-Dickstein, J. (2017). Learned optimizers that scale and generalize. In *International Conference on Machine Learning*. (pp. 27, 81, 85, 87, and 88.)

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*. (pp. 10, 14, and 101.)

Wurman, P. R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K., Walsh, T. J., Capobianco, R., Devlic, A., Eckert, F., Fuchs, F., et al. (2022). Outracing champion gran turismo drivers with deep reinforcement learning. *Nature*. (p. 1.)

Xu, M., Quiroz, M., Kohn, R., and Sisson, S. A. (2019). Variance reduction properties of the reparameterization trick. In *The 22nd International Conference on Artificial Intelligence and Statistics*. (pp. 16 and 108.)

Xu, Z., van Hasselt, H. P., Hessel, M., Oh, J., Singh, S., and Silver, D. (2020). Meta-gradient reinforcement learning with an objective discovered online. *Advances in Neural Information Processing Systems*. (p. 83.)

Xu, Z., van Hasselt, H. P., and Silver, D. (2018). Meta-gradient reinforcement learning. *Advances in neural information processing systems*. (p. 28.)

Yarotsky, D. (2017). Error bounds for approximations with deep ReLU networks. *Neural Networks*. (p. 91.)

Yoon, J., Yang, E., Lee, J., and Hwang, S. J. (2018). Lifelong learning with dynamically expandable networks. In *International Conference on Learning Representations*. (pp. 34 and 62.)

You, S., Xu, C., Xu, C., and Tao, D. (2017). Learning from multiple teacher networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. (p. 27.)

Young, K. and Tian, T. (2019). MinAtar: An Atari-inspired testbed for more efficient reinforcement learning experiments. *arXiv preprint arXiv:1903.03176*. (p. 47.)

Young, K., Wang, B., and Taylor, M. E. (2019). Metatrace actor-critic: Online step-size tuning by meta-gradient descent for reinforcement learning control. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. (p. 28.)

Zagoruyko, S. and Komodakis, N. (2017). Paying more attention to attention: Improving the

performance of convolutional neural networks via attention transfer. In *International Conference on Learning Representations*. (p. 27.)

Zeng, G., Chen, Y., Cui, B., and Yu, S. (2019). Continual learning of context-dependent processing in neural networks. *Nature Machine Intelligence*. (p. 34.)

Zenke, F., Poole, B., and Ganguli, S. (2017). Continual learning through synaptic intelligence. In *International Conference on Machine Learning*. (pp. 34 and 35.)

Zha, D., Xie, J., Ma, W., Zhang, S., Lian, X., Hu, X., and Liu, J. (2021). DouZero: Mastering DouDizhu with self-play deep reinforcement learning. *International Conference on Machine Learning*. (pp. 1 and 100.)

Zhang, L., Song, J., Gao, A., Chen, J., Bao, C., and Ma, K. (2019a). Be your own teacher: Improve the performance of convolutional neural networks via self distillation. In *Proceedings of the IEEE/CVF international conference on computer vision*. (p. 27.)

Zhang, S. (2018). Modularized implementation of deep RL algorithms in PyTorch. https://github.com/ShangtongZhang/DeepRL. (p. 113.)

Zhang, S., Laroche, R., van Seijen, H., Whiteson, S., and Tachet des Combes, R. (2022). A deeper look at discounting mismatch in actor-critic algorithms. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*. (p. 13.)

Zhang, S., Yao, L., Sun, A., and Tay, Y. (2019b). Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys*. (p. 1.)

Zhang, Y., Khanduri, P., Tsaknakis, I., Yao, Y., Hong, M., and Liu, S. (2024). An introduction to bilevel optimization: Foundations and applications in signal processing and machine learning. *IEEE Signal Processing Magazine*. (p. 27.)

Zhao, W.-Y., Guan, X.-Y., Liu, Y., Zhao, X., and Peng, J. (2019). Stochastic variance reduction for deep q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. (p. 2.)

Zhao, X., Xia, L., Zhang, L., Ding, Z., Yin, D., and Tang, J. (2018). Deep reinforcement learning for page-wise recommendations. In *Proceedings of the 12th ACM Conference on Recommender Systems*. (pp. 1 and 100.)

Zheng, G., Zhang, F., Zheng, Z., Xiang, Y., Yuan, N. J., Xie, X., and Li, Z. (2018). DRN: A deep reinforcement learning framework for news recommendation. In *Proceedings of the 2018 World Wide Web Conference*. (pp. 1 and 100.)

Zhou, D., Ye, M., Chen, C., Meng, T., Tan, M., Song, X., Le, Q., Liu, Q., and Schuurmans, D. (2020). Go wide, then narrow: Efficient training of deep thin networks. In *International Conference on Machine Learning*. (p. 63.)