International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 10 Issue 8 ||August 2022 ||PP. 187-321

Addressing Challenges in Deep Reinforcement Learning: Efficient Sampling of Actions, States, and Trajectories

Junxia Deng
University of Southern California
California, USA

Ceil Hong. Zhang
Massachusetts Institute of Technology
Massachusetts, USA
zhanghongceil@pku.org.cn

Abstract

The combination of deep neural networks with reinforcement learning (RL) shows great promise for solving otherwise intractable learning tasks. However, practical demonstrations of deep reinforcement learning remain scarce. The challenges in using deep RL for a given task can be grouped into two categories, broadly "What to learn from experience?" and "What experience to learn from?" In this thesis, I describe work to address the second category. Specifically, problems of sampling actions, states, and trajectories which contain information relevant to learning tasks. I examine this challenge at three levels of algorithm design and task complexity, from algorithmic components to hybrid combination algorithms that break common RL conventions.

In the first chapter, I describe work on stable and efficient sampling of actions that optimize a Q-function of continuous-valued actions. By combining a sample-based optimizer with neural network approximation, it is possible to obtain stability in training, computational efficiency, and precise inference.

In the second chapter, I describe work on reward-aware exploration, the discovery of desirable behaviors where common sampling methods are insufficient. A teacher "exploration" agent discovers states and trajectories which maximize the amount a student "exploitation" agent learns on those experiences, and can enable the student agent to solve hard tasks which are otherwise impossible.

In the third chapter, I describe work combining reinforcement learning with heuristic search, for use in task domains where the transition model is known, but where the combinatorics of the state space are intractable for traditional search. By combining deep Q-learning with a best-first tree search algorithm, it is possible to find solutions to program synthesis problems with dramatically fewer samples than with common search algorithms or RL alone.

Lastly, I conclude with a summary of the major takeaways of this work, and discuss extensions and future directions for efficient sampling in RL.

Contents

Listing of figures

0 Introduction

Broadly put, reinforcement learning (RL) is a field that studies how intelligent agents can interact with an environment about which the agent knows and assumes little. Typically, the only assumptions the agent can make about the Markov Decision Process (MDP)[104] it seeks to solve are that the (partially observable) state of the MDP changes over time, that the agent can affect the state through its actions, and that there exists a reward function of state which the agent should maximize. Core questions such what a rewarding state looks like, what aspects of the environment the agent can influence, or what the effect of a given action will be must be discovered by the agent. Without interacting with the environment to explore it and gather data, the agent cannot achieve its intended task [*].

At a high level, this problem of learning about the environment can be broken down into two sub-problems– First, the problem of learning from the data the agent has collected, which is most directly affected by modifications to the agent's learning rule as well as learning agent architecture and parametrization, and second the problem of collecting data to be learned from. Most work in the field of reinforcement learning is concerned with the first of these, but the second is increasingly seen as key to success in many major application domains of RL.

While this second problem encompasses many different subproblems such as exploration, offline RL, and action selection, the overarching focus is on data, rather than learning algorithms. In the context of deep RL, there is reason for hope in this focus on data– Multiple fields of artificial intelligence have found that using large and deep neural network function approximators with large (and well-crafted) datasets can yield impressive results. While the nature of

[*]There are of course algorithms which break these assumptions, including the focus of Chapter 3 and others which are discussed briefly in Chapter 4

the “data” problems in RL differ from those of supervised learning, it is thus reasonable to think that intelligent selection of training data in deep RL could also yield significant gains over naive data collection methods. In this thesis, I will focus on problems of this sort, in particular those that can be collectively described as “sampling” problems.

0.1 Sampling in Reinforcement Learning

So what is “sampling” in RL? Broadly put, it is the problem of collecting interactions with the environment MDP that allow an RL agent to learn to solve it. An RL agent must learn to perform a given task without prior knowledge of what that task entails, and to do this it must learn through a trial-and-error process of sampling attempts to perform the task and learning which actions lead to success and which lead to failure from that data.

Contained in this problem statement is an unanswered question– What attempts does the agent sample? Indeed, this question is fundamental to any learning agent trained on data, and its expressions in other fields of machine learning (dataset composition in supervised learning, training objective and regimen in self-supervised learning) have proven to make a large difference in the performance for many subproblems. In contrast, this question has less effective and less general answers in RL, with some exceptions (as we will see). Given this significance, the core focus of this thesis will be on finding answers to the sampling question.

Theoretically, standard RL algorithms will converge to solve a task correctly assuming infinite samples where every possible action is sampled with nonzero probability[104]. While complete, this answer is obviously unsatisfying– infinite time convergence guarantees are both theoretically and practically unsatisfying. However, to make a stronger guarantee we must make additional assumptions about the problem structure. These assumptions can be practical or impractical, and depend heavily on the problem domain in question– they are a key factor in the design of a practical RL system for a given task. Indeed, some of the most notably success stories in RL, such as superhuman performance on the game of Go[97], are made possible by clever task design, while in other application domains such as robotics RL struggles to gain traction despite considerable research efforts.

Let's consider a couple of case studies:

0.2 Case Study: AlphaGo Zero

A seminal paper from Silver et al., AlphaGo Zero learned to play the board game Go at a superhuman level, entirely through self-play with no human-produced training data. Examining the training curves from that paper (see Figure 1) reveals a classic asymptotic training curve, with rapid improvement early in training followed by a gradual decline in the rate of policy improvement later on. This curve shape is indicative of a task where the sampling problem is solved relatively well– early on, when the agent knows very little about the game of Go

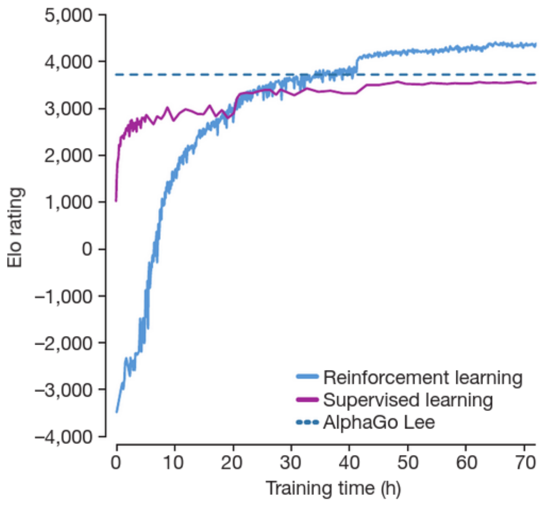


Figure 1: Training curves for AlphaGo Zero. Performance improves smoothly and asymptotically throughout training.

it still samples data that allows for rapid improvement, while later in training performance continues to smoothly improve up to an extremely high level.

Obtaining such an ideal result on Go is a function of many different factors– the architecture of the neural networks used, the RL learning algorithm, the fully observable game state, and more. However, the same algorithms and neural networks can be applied to another game (for example, StarCraft[110]) and perform poorly due to how the structure of the game enables efficient sampling.

Why? Go is a game with two players, where one player must lose and the other win. Due to the size of the board and the number of stones, this is guaranteed to happen within a reasonable maximum length, and every move prior to the end of the game contributes to the outcome– there are few “pointless” turns in Go. By playing against copies of itself, an agent, whether near-random or su-

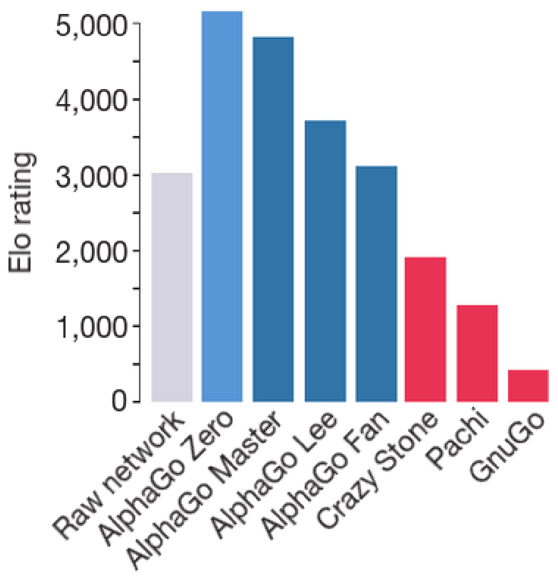


Figure 2: Performance of the raw AlphaGo Zero policy with ("AlphaGo Zero") and without ("Raw network") MCTS to select moves.

perhuman, will play against a similarly skilled opponent, and thus should win 50% of the time. At random, the agent may play better than typical or worse than typical and win more or less often, which produces a gradient for the policy towards better plays and away from worse ones. This happens regardless of the agent's skill level, resulting in a smooth asymptotic training curve as the agent samples experiences that help it improve throughout training. Because Go has a structure that makes sampling good training data easy, training is relatively fast given the complexity of the game and asymptotic performance is extremely good. Efficient sampling makes superhuman performance tractable to obtain.

It is worth noting that sampling techniques can improve performance even in a game like Go where efficient sampling is easy to obtain. In AlphaGo Zero, the authors use Monte Carlo Tree Search (MCTS) to select actions based on accumulated statistics, which as shown in Figure 2 improves performance significantly over the baseline of simply using the policy's predicted actions. While the raw network is high skilled at Go, MCTS sampling allows for better action selection at inference time.

0.3 Case Study: Qt-Opt

While the game of Go has structure conducive to efficient sampling, many other tasks of human interest do not. A major example is the field of robotics writ large. Robots capable of complex human-like manipulation tasks would be a great benefit to society, and much work has been put into developing such robots, but to date only limited success has been found, and RL-based approaches in particular remain largely experimental.

The reasons for this largely come down to task structure, and how it effects sampling. Robotic simulators have limited accuracy for many types of interactions[19], and as such a robotic RL agent must be trained in the real world, where time spent sampling experiences is precious. Unlike AlphaGo Zero, where millions of games of Go could be played and trained on in a day, robots must collect data in real time, and efficient data collection is therefore critical to success. Complicating this is the discontinuous structure of most robotic tasks– con-

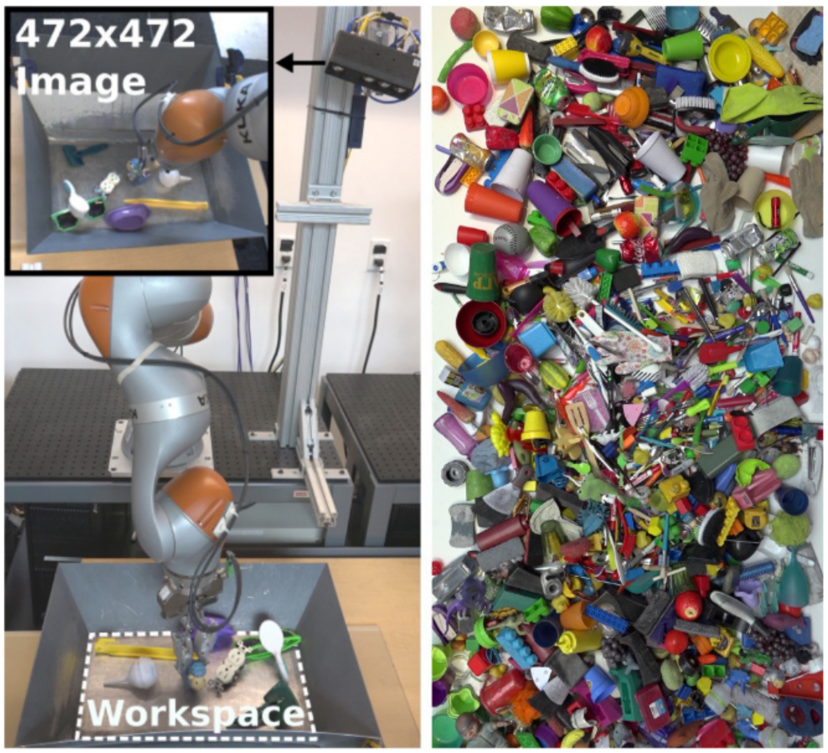


Figure 3: Picture of the Qt-Opt robotic system (left) and the objects Qt-Opt learns to grasp (right).

sider a robot arm attempting to pick up objects. With a few corner cases, the robot will either succeed (the object has been grasped and lifted), or fail (the object remains on the ground). As very few robot trajectories result in a successful grasp, an untrained RL policy will sample very inefficiently– it may be hundreds or thousands of trials before it manages to grasp the object at all to learn anything about the task. Structuring a robotic task similar to Go where the agent will automatically succeed 50% of the time is difficult or impossible in most cases, and considerable manual engineering work is required to structure a task such that it is learnable from scratch by an RL agent.

To illustrate this, let's look at an example case study. Qt-Opt[55] is seminal robotic systems paper detailing a robot capable of picking miscellaneous objects out of a bin using RL trained entirely on real-world data, among the more complex real-world robotic tasks accomplished by an RL agent to date. To make this tractable, a number of major engineering challenges had to be overcome:

- The space of the task was constrained, with a limited working volume and short time horizons (max 30 actions), so that a weak policy will succeed more often.
- Qt-Opt used data collected by a heuristic policy to bootstrap training and give the RL policy some examples of success to train on.
- The neural networks used to control the robot are large and expensive to evaluate, and the system requires a compute cluster and does not run in real time.
- Most notably, the authors used a large number of robots to sample data in parallel, collecting a dataset of 580,000 grasp attempts in 800 robot-hours, or just over one robot-month.

While the results are impressive, this approach requires considerable task-specific engineering and millions of dollars of robot hardware. Further, research suggests that while generalization to new objects is fairly good, such a system will not generalize to small changes in environment, such as changing the color of the robot arm, with fine-tuning requiring hundreds of thousands more grasp attempts to adapt to small changes[65]. As practical robots must be robust to large changes in their work environment without months of robot time spent retraining, this approach remains impractical for most applications.

To improve on this, there are two options. If the system were to generalize better given the same amount of data, spending the money and time required to collect that data is reasonable to do once or twice. Conversely, if similar performance could be obtained using less data (for example, 5000 grasps rather than nearly 500,000) it would be tractable to collect training data in diverse environments and retrain as needed when conditions change. In both cases, the challenge can be further decomposed into improving how the data is used (better generalizing neural net architectures, more efficient training from small amounts of data) and improving what data is collected (target data collection such that the 500k grasps cover the task space broadly, or so that the 5k grasps are efficiently selected for learning the task quickly).

0.4 Contributions

As we can see from the case studies in the previous section, the difficulty of sampling good data for RL varies tremendously depending on the task domain. Further, the reasons for that difficulty also differ. While general purpose techniques are desirable, this heterogeneity encourages a domain-specific approach. In each chapter of this thesis, I will outline a domain in which sampling is challenging, and propose an algorithm to address that challenge. While these algorithms are designed with a specific domain in mind, many of their principles can be applied more broadly.

The body of this thesis consists of the following sections:

- Chapter 1: **Sampling Stable Optimal Actions**

  In this chapter, I address the challenge of sampling actions from a Q-function of a continuous action vector $a \in \mathcal{R}^N$. This is hard to do because neural network Q-functions are highly non-linear, so optimizing their input is a non-convex optimization problem that must be performed every time one wants to sample actions or even train the Q-function. Previous work demonstrated that using sample-based optimization algorithms such as the cross-entropy method results in good stability for training. However, this is computationally slow due to the serial nature of the algorithm and need to evaluate the Q-function many times per iteration. I extend this approach by using supervised learning to train a small neural network policy that approximates the output of the sample-based optimizer with much less computation required. The result is a "best of both worlds" algorithm that is stable to train while being fast at inference time.

- Chapter 2: **Exploration via Reward Prediction Error**

  Following from the previous chapter, here I introduce the problem of exploration, sampling longer sequences of actions which yield trajectories containing useful information for learning the given task. While in the limit this problem is NP-hard (it becomes necessary to visit every sequence of states to discover the singular correct trajectory), in practice other information can often be used to prioritize some states over others. Much work has focused on sampling novel states (or observations in the partially-observable case), with varying definitions of what constitutes a novel state. Here, I introduce an alternate exploration objective, the temporal difference (TD) error of a Q-function, which focuses exploration on states that affect the agent's expected returns. To do this, I train one Q-function to estimate the TD-error of a second Q-function, then roll out the first Q-function in the task to discover state-actions that result in high TD-error for the second. The result is an exploration algorithm that generalizes to many types of exploration task, including those where state novelty is not well correlated with trajectories that result in improved returns.

- Chapter 3: **Q-Function Prioritized Tree Search**

  In this chapter, I extend the problem of exploration to a radically different domain, program synthesis. In this task, the environment consists of a finite but intractably large number of states (possible programs), a large number of possible actions (modifications to the current program), and a perfect world model (programs are modular and can be executed). However, due to the nature of the subspace of human-relevant programs, the reward optimization landscape is nearly pathological, consisting of bottleneck states, saddle points and plateaus of similarly-rewarding programs, and optimal programs can be nearly singular in reward space. To address this extreme sampling challenge, I introduce Reinforcement Learning

Guided Tree Search (RLGTS), a hybrid algorithm that uses a Q-function to prioritize candidate programs and a best-first tree search on the Q-scored programs to sample the finite-but-intractable space of possible programs that might satisfy a given specification. This algorithm departs from some assumptions of typical MDPs, such as episodic structure or sampling timesteps linearly, and by doing so achieves orders of magnitude better sample efficiency than either traditional tree search algorithms or traditional reinforcement learning on simple programs of floating point arithmetic.

Following these sections, I then conclude the thesis with some reflections on these three projects and the lessons learned about sampling. I also discuss some future directions regarding sampling, and discuss some recent work in promising directions.

1

Sampling Stable Optimal Actions

1.1 Introduction

In recent years, model-free deep reinforcement learning (RL) algorithms have demonstrated the capacity to learn sophisticated behavior in complex environments. Starting with Deep Q-Networks (DQN) achieving human-level performance on Atari games[73], deep RL has led to impressive results in several classes of challenging tasks. While many deep RL methods were initially limited to discrete action spaces, there has since been substantial interest in applying deep RL to continuous action domains. In particular, deep RL has increasingly been studied for use in continuous control problems, both in simulated environments and on robotic systems in the real world.

A number of challenges exist for practical control tasks such as robotics. For tasks involving a physical robot where on-robot training is desired, the physical constraints of robotic data collection render data acquisition costly and time-consuming. Thus, the use of off-policy methods like Q-learning is a practical necessity, as data collected during development or by human demonstrators can be used to train the final system, and data can be re-used during training. However, even when using off-policy Q-learning methods for continuous control, several other challenges remain. In particular, training stability across random seeds, hyperparameter sensitivity, and runtime are all challenges that are both relatively understudied and are critically important for practical use.

Inconsistency across runs, e.g. due to different random initializations, is a major issue in many domains of deep RL, as it makes it difficult to debug and evaluate an RL system. Deep Deterministic Policy Gradients (DDPG), a popular off-policy Q-learning method[66], has been repeatedly characterized as unstable[22,54]. While some recent work has improved stability in off-policy Q-learning, there remains significant room for improvement[43,45,34]. Sensitivity to hyperparameters (i.e. batch size, network architecture, learning rate, etc) is a particularly critical issue when system evaluation is expensive, since debugging and task-specific tuning are difficult and time consuming to perform. Finally, many real robotics tasks have strict runtime and hardware constraints (i.e. interacting with a dynamic system), and any RL control method applied to these tasks must be fast enough to compute in real time.

Mitigating these challenges is thus an important step in making deep RL practical for continuous control. In this chapter, I will introduce Cross-Entropy Guided Policy (CGP) learning, a general Q-function and policy training method that can be combined with most deep Q-learning methods and demonstrates improved stability of training across runs, hyperparameter combinations, and tasks, while avoiding the computational expense of a sample-based policy at inference time. CGP is a multi-stage algorithm that learns a Q-function using a heuristic Cross-Entropy Method (CEM) sampling policy to sample actions, while training a deterministic neural network policy in parallel to imitate the CEM policy. This learned policy is then used at inference time for fast and precise evaluation without expensive sample iteration. I will show that this method achieves performance comparable to state-of-the-art methods on standard continuous-control benchmark tasks, while being more robust to hyperparameter tuning and displaying lower variance across training runs. Further, I will show that its inference-time runtime complexity is 3-6 times better than when using the CEM policy for inference, while slightly outperforming the CEM policy. This combination of attributes (reliable training and cheap inference) makes CGP well suited for real-world robotics tasks and other time/compute sensitive applications.

1.2 Related Work

The challenge of reinforcement learning in continuous action spaces has been long studied[96,47], with recent work building upon *on-policy* policy gradient methods[103] as well as the *off-policy* deterministic policy gradients algorithm[96]. In addition to classic policy gradient algorithms such as REINFORCE[103] or Advantage Actor Critic[20], a number of recent on-policy methods such as TRPO[91] and PPO[92] have been applied successfully in continuous-action domains, but their poor sample complexity makes them unsuitable for many real world applications, such as robotic control, where data collection is expensive and complex. While several recent works[69,117,6] have successfully used simulation-to-real transfer to train in simulations where data collection is cheap, this process remains highly application-specific, and is difficult to use for more complex tasks.

Off-policy Q-learning methods have been proposed as a more data efficient alternative, typified by Deep Deterministic Policy Gradients (DDPG)[66]. DDPG trains a Q-function similar to Mnih et al., while in parallel training a deterministic policy function to sample good actions from the Q-function. Exploration is then achieved by sampling actions in a noisy way during policy rollouts, followed by off-policy training of both Q-function and policy from a replay buffer. While DDPG has been used to learn non-trivial policies on many tasks and benchmarks[66], the algorithm is known to be sensitive to hyperparameter tuning and to have relatively high variance between different random seeds for a given configuration[22,50]. Recently multiple extensions to DDPG have been proposed to improve performance, most notably Twin Delayed Deep Deterministic Policy Gradients (TD3)[34] and Soft Q-Learning (SQL)/Soft Actor-Critic (SAC)[43,44].

TD3 proposes several additions to the DDPG algorithm to reduce function approximation error: it adds a second Q-function to prevent over-estimation bias from being propagated through the target Q-values and injects noise into the target actions used for Q-function bootstrapping to improve Q-function smoothness. The resulting algorithm achieves significantly improved performance relative to DDPG, and I use their improvements to the Q-function training algorithm as a baseline for CGP.

In parallel with TD3,[44] proposed Soft Actor Critic as a way of improving on DDPG's robustness and performance by using an entropy term to regularize the Q-function and the reparametrization trick to stochastically sample the Q-function, as opposed to DDPG and TD3's deterministic policy. SAC and the closely related Soft Q-Learning (SQL)[43] have been applied successfully for real-world robotics tasks[45,42].

Several other recent works propose methods that use CEM and stochastic sampling in RL. Evolutionary algorithms take a purely sample-based approach to fitting a policy, including fitting the weights of neural networks, such as in Salimans et al., and can be very stable to train, but suffer from very high computational cost to train. Evolutionary Reinforcement Learning (ERL)[59] combines evolutionary and RL algorithms to stabilize RL training. CEM-RL[81] uses CEM to sample populations of policies which seek to optimize actions for a Q-function trained via RL, while I optimize the Q-function actions directly via CEM sampling similar to Qt-Opt[56].

There exists other recent work that aims to treat learning a policy as supervised learning[2,1,116]. Abdolmaleki et al. propose a formulation of policy iteration that samples actions from a stochastic learned policy, then defines a locally optimized action probability distribution based on Q-function evaluations, which is used as a target for the policy to learn[2,1].

The baseline for my method is modeled after the CEM method used in the Qt-Opt system, a method described by Kalashnikov et al. for vision-based dynamic manipulation trained mostly off-policy on real robot data. Qt-Opt eschews the use of a policy network as in most other continuous-action Q-learning methods, and instead uses CEM to directly sample actions that are optimal with respect to the Q-function for both inference rollouts and training. They describe the method as being stable to train, particularly on off-policy data, and demonstrate its usefulness on a challenging robotics task, but do not report its performance on standard benchmark tasks or against other RL methods for continuous control. I base my CEM sampling of optimal actions on their work, generalized to MuJoCo benchmark tasks, and extend it by learning a deterministic policy for use at inference time to improve performance and computational complexity, avoiding the major drawback of the method- the need to perform expensive CEM sampling for every action at inference time (which must be performed in real time on robotic hardware).

1.3 Preliminaries

I describe here the notation of the continuous-control RL domain, based on the notation defined by Sutton & Barto. Reinforcement learning is a class of algorithms for solving Markov Decision Problems (MDPs), typically phrased in the finite time horizon case as an agent characterized by a policy π taking actions a_t in an environment, with the objective of maximizing the expected total reward value $\mathbb{E}\sum_{t=1}^{T}\gamma^t r(s_t, a_t)$ that agent receives over timesteps $t \in \{1 \dots T\}$ with some time decay factor per timestep γ. To achieve this, we thus seek to find an optimal policy π^* that maximizes the following function:

$$J(\pi) = \mathbb{E}_{s,a\sim\pi}[\sum_{t=1}^{T}\gamma^t r(s_t, a_t)]$$

A popular class of algorithms for solving this is Q-learning, which attempts to find an optimal policy by finding a function

$$Q^*(s_t, a_t) = r(s_t, a_t) + \gamma \text{max}_{a_{t+1}}(Q^*(s_{t+1}, a_{t+1}))$$

which satisfies the Bellman equation[104]:

$$Q(s, a) = r(s, a) + \mathbb{E}[Q(s', a')], \quad a' \sim \pi^*(s')$$

Once Q^* is known π^* can easily be defined as $\pi^*(s) = \text{argmax}_a(Q^*(s, a))$. Q-learning attempts to learn a function Q_θ that converges to Q^*, where θ is the parameters to a neural network. Q_θ is often learned through bootstrapping, wherein we seek to minimize the function

$$J(\theta) = \mathbb{E}_{s,a}[(Q_\theta - [r(s,a) + \gamma \text{max}_{a'}(\hat{Q}(s', a'))])^2]$$

where $\hat{Q}$ is a target Q-function, here assumed to be a time delayed version of the current Q-function, $\hat{Q}_{\hat{\theta}}$[74].

To use the above equation, it is necessary to define a function $\pi(s)$ which computes $\text{argmax}_a(Q(s,a))$. In discrete action spaces, $\pi(s)$ is trivial, since argmax_a can be computed exactly by evaluating each possible a with Q. In continuous-valued action spaces, such a computation is intractable. Further, as most neural network Q-functions are highly non-convex, an analytical solution is unlikely to exist. Various approaches to solving this optimization problem have been proposed, which have been shown to work well empirically. Lillicrap et al. show that a neural network function for sampling actions that approximately maximize the Q-function can be learned using gradients from the Q-function. This approach forms the basis of much recent work on continuous action space Q-learning.

1.4 From Sampling-based Q-learning to Cross-Entropy Guided Policies (CGP)

In this section, I will first describe an established method for using a sampling-based optimizer to optimize inputs to a Q-function which can be used as a pol-

Algorithm 1 Cross Entropy Method Policy (π_{CEM}) for Q-Learning

```
Input: state $s$, Q-function $Q$, iterations $N$, samples $n$, winners $k$, action dimension $d$
$\boldsymbol{\mu} \leftarrow \mathbf{0}^d$
$\boldsymbol{\sigma}^2 \leftarrow \mathbf{1}^d$
for $t = 1$ to $N$ do
    $A \leftarrow \{\boldsymbol{a_i} : \boldsymbol{a_i} \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)\}$
    $\tilde{A} \leftarrow \{\tilde{\boldsymbol{a}}_{\boldsymbol{i}} : \tilde{\boldsymbol{a}}_{\boldsymbol{i}} = \tanh(\boldsymbol{a_i})\}$
    $\mathcal{Q} \leftarrow \{q_i : q_i = Q(\tilde{\boldsymbol{a}}_{\boldsymbol{i}})\}$
    $I \leftarrow \{\text{sort}(\mathcal{Q})_i : i \in [1, \ldots, k]\}$
    $\boldsymbol{\mu} \leftarrow \frac{1}{k} \sum_{i \in I} \boldsymbol{a_i}$
    $\hat{\boldsymbol{\sigma}}^{\mathbf{2}} \leftarrow \text{Var}_{i \in I}(\boldsymbol{a_i})$
    $\boldsymbol{\sigma}^{\mathbf{2}} \leftarrow \hat{\boldsymbol{\sigma}}^{\mathbf{2}}$
end for
return $\tilde{\boldsymbol{a}}^* \in \tilde{A}$ such that $Q(\tilde{\boldsymbol{a}}^*) = \max_{i \in I} Q(\tilde{\boldsymbol{a}}_{\boldsymbol{i}})$
```

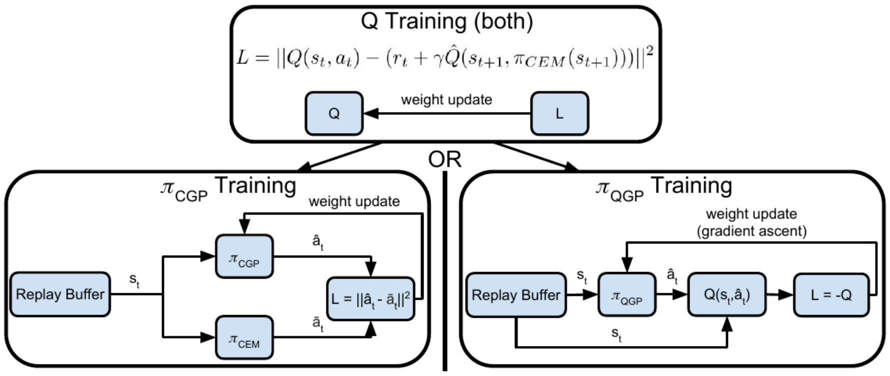


Figure 1.1: Schematic diagram of CGP and QGP. Both CGP and QGP use the same training method to train their Q-functions. However CGP (left) regresses π_{CGP} on the L2-norm between the current π_{CGP} and the CEM-based policy π_{CEM}, while QGP (right) trains π_{QGP} to maximize Q given s_t by directly performing gradient ascent on Q.

icy to train the Q-function via standard Q-learning. I will then present two novel methods for training deterministic policies separately from the Q-function.

1.4.1 Q-Learning with Sampling-Based Policies

The basis for my method is the use of a sampling-based optimizer to compute approximately optimal actions with respect to a given Q function and a given state s. Formally, I define the policy $\pi_{S_Q}(s) = S_Q(s)$, where S_Q is a sampling-based optimizer that approximates $\text{argmax}_a Q(s, a)$ for action a and state observation s. I can then train a Q-function Q_θ parameterized by the weights of a neural network using standard Q-learning as described in Section 1.3 to minimize:

$$J(\theta) = \mathbb{E}_{s,a}[(Q_\theta - [r(s, a) + \gamma\hat{Q}(s', \pi_{S_{\hat{Q}_\theta}}(s'))])^2]$$

The choice of sampling-based optimizer S_Q can have a significant impact on the quality of the policy it induces, and therefore has a significant impact on the quality of Q_θ after training - while I leave the exploration of optimal sampling methods to future work, I used a simple instantiation of the Cross-Entropy method (CEM), which was empirically demonstrated by Kalashnikov et al. to work well for certain continuous-control tasks[56]. In this formulation, each action vector is represented as a collection of independent Gaussian distributions, initially with mean $\mu = 1$ and standard deviation $\sigma = 1$. These variables are sampled n times to produce action vectors $a_0, a_1, ..., a_{n-1}$, which are then scored by Q. The top k scoring action vectors are then used to reparameterize the Gaussian distributions, and this process is repeated N times. For brevity, I refer to this policy as π_{CEM}. The full algorithm can be found in Algorithm 1.

1.4.2 Imitating π_{CEM} with a Deterministic Policy

While π_{CEM} is competitive with learned policies for sampling the Q-function (described in Section 1.5), it suffers from poor runtime efficiency, as evaluating many sampled actions is computationally expensive, especially for large neural networks. Further, there is no guarantee that sampled actions will lie in a local minimum of the Q-value energy landscape due to stochastic noise. My main methodological contribution in this work, formalized in Algorithm 2, is the extension of π_{CEM} by training a deterministic neural network policy $\pi_\phi(s)$ to predict an approximately optimal action at inference time, while using π_{CEM} to sample training data from the environment and to select bootstrap actions for training the Q-function.

A single evaluation of π_ϕ is much less expensive to compute than the multiple iterations of Q-function evaluations required by π_{CEM}. Even when evaluating CEM samples with unbounded parallel compute capacity, the nature of iterative sampling imposes a serial bottleneck that means the theoretical best-case runtime performance of π_{CEM} will be N times slower than π_ϕ. Additionally, as π_{CEM} is inherently noisy, by training π_ϕ on many approximately optimal actions from $\pi_{\text{CEM}}(s)$ evaluated on states from the replay buffer, I expect that, for a given state s and Q_θ, π_ϕ will converge to the mean of the samples from π_{CEM}, reducing policy noise at inference time.

While the idea of training an inference-time policy to predict optimal actions with respect to Q_θ is simple, there are several plausible methods for training π_ϕ. I explore four related methods for training π_ϕ, the performance of which are discussed in Section 1.5. The high-level differences between these methods can be found in Figure 1.1.

Q-gradient-Guided Policy

A straightforward approach to learning π_ϕ is to use the same objective as DDPG[66]:

$$J(\phi) = \mathbb{E}_{s\sim\rho^{\pi_{\text{CEM}}}}(\nabla_{\pi_\phi} Q_\theta(s, \pi_\phi(s)))$$

to optimize the weights ϕ off-policy using Q_θ and the replay data collected by π_{CEM}. This is the gradient of the policy with respect to the Q-value, and for an optimal Q should converge to an optimal policy. Since the learned policy is not used during the training of the Q-function, but uses gradients from Q to learn an optimal policy, I refer to this configuration as Q-gradient Guided Policies (QGP), and refer to policies trained in this fashion as π_{QGP}. I tested two versions of this method, an "offline" version where π_ϕ is trained to convergence on a fixed Q-function and replay buffer, and an "online" version where π_ϕ is trained in parallel with the Q-function, analogous to DDPG other than that π_ϕ is not used to sample the environment or to select actions for Q-function bootstrap targets. I refer to these variants as QGP-Offline and QGP-Online respectively.

Cross-Entropy-Guided Policy

However, as shown in Figure 1.3, while both variants that train π_ϕ using the gradient of Q_θ can achieve good performance, their performance varies significantly depending on hyperparameters, and convergence to an optimal (or even good) policy does not always occur. I hypothesize that the non-convex nature of Q_θ makes off-policy learning somewhat brittle, particularly in the offline case, where gradient ascent on a static Q-function is prone to overfitting to local maxima. I therefore introduce a second variant, the Cross-Entropy Guided Policy (CGP), which trains π_ϕ using an L2 regression objective

$$J(\phi) = \mathbb{E}_{s_t \sim \rho^{\pi_{\text{CEM}}}}(\nabla_{\pi_\phi} ||\pi_\phi(s_t) - \pi_{\text{CEM}}(s_t)||^2)$$

This objective trains π_ϕ to imitate the output of π_{CEM} without relying on CEM for sampling or the availability of Q_θ at inference time. If I assume π_{CEM} is an approximately optimal policy for a given Q_θ (an assumption supported by my empirical results in Section 1.5), this objective should converge to the global maxima of Q_θ, and avoids the local maxima issue seen in QGP. As π_{CEM} can only be an approximately optimal policy, CGP may in theory perform worse than QGP since QGP optimizes Q_θ directly, but I show that this theoretical gap does not result in diminished performance. Moreover, I demonstrate that CGP is significantly more robust than QGP, especially in the offline case. I explore both online and offline versions of this method similar to those described for QGP.

While QGP and CGP are compatible with any Q-learning algorithm, to improve performance and training stability further I combine them with the TD3 Q-learning objective described in Fujimoto et al., which adds a second Q-function for target Q-value computation to minimize function approximation error, among other enhancements. My method of using π_{CEM} to sample actions for Q-function training and training π_ϕ for use at inference time is agnostic to the form of the Q-function and how it is trained, and could be combined with future Q-learning methods. Pseudocode for the full CGP method can be found in Algorithm 2.

1.5 Experiments

To characterize my method, I conducted a number of experiments in various simulated environments.

1.5.1 Experiment Setup

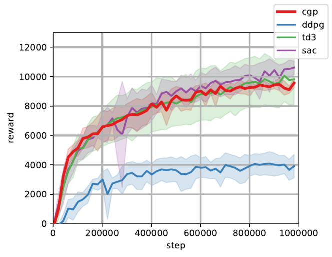
My experiments are intended to highlight differences between the performance of CGP and current state-of-the-art methods on standard RL benchmarks. I compare against DDPG, TD3, Soft Actor-Critic (SAC), and an ablation of my method which does not train a deterministic policy but instead simply uses π_{CEM} to sample at test time similar to the method of Kalashnikov et al.. To obtain consistency across methods and with prior work I used the author's publicly available implementations for TD3 and SAC, but within my own training framework to ensure consistency. I attempt to characterize the behavior of these methods across multiple dimensions, including maximum final reward achieved given well-tuned hyperparameters, the robustness of the final reward across diverse hyperparameters, the stability of runs within a given hyperparameter set, and the inference time computational complexity of the method.

I assessed my method on an array of continuous control tasks in the MuJoCo simulator through the OpenAI gym interface, including `HalfCheetah`, `Humanoid`, `Ant`, `Hopper`, `Pusher`[11]. These tasks are intended to provide a range of complexity, the hardest of which require significant computation in order to achieve good results. The dimensionality of the action space ranges from 2 to 17 and the state space 8 to 376. Because of the large amount of computation required to train on these difficult tasks, robustness to hyperparameters is extremely valuable, as the cost to exploring in this space is high. For similar reasons, stability and determinism limit the number of repeated experiments required to achieve an estimate of the performance of an algorithm with a given degree of certainty. In order to test robustness to hyperparameters, I choose one environment (`HalfCheetah`) and compare CGP with other methods under a sweep across common hyperparameters. To test stability, I performed 4 runs with unique random seeds for each hyperparameter combination. Each task is run for 1 million time steps, with evaluations every 1e4 time steps. Additional hyperparameters are described in Appendix A.1.

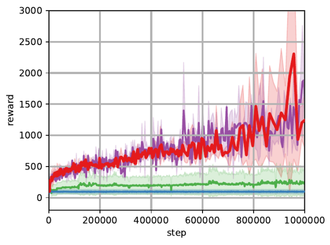
After tuning hyperparameters on `HalfCheetah`, I then selected a single common "default" configuration that worked well on each method, the results of which for `HalfCheetah` are shown in Figure 1.2. I then ran this configuration on each other benchmark task, as a form of holdout testing to see how well a generic set of hyperparameters will do for unseen tasks. Additional experimental and implementation details can be found in Appendix A.3.

I also evaluated several variants of my method, as described in Section 1.4.2. I compare robustness and peak performance for both online and offline versions of CGP and QGP.

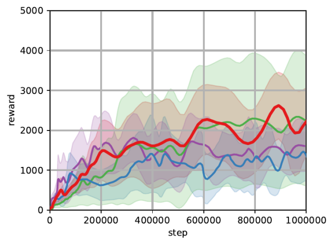
1.5.2 Comparisons



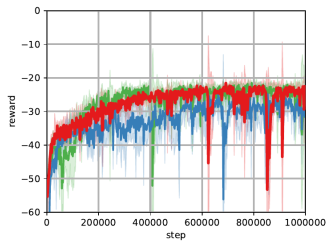
(a) HalfCheetah



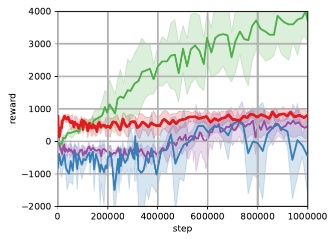
(b) Humanoid



(c) Hopper



(d) Pusher



(e) Ant

Figure 1.2: Performance of various methods (CGP, SAC, DDPG, and TD3) on OpenAI Gym benchmark tasks, simulated in MuJoCo. In all cases CGP is either the best or second best performing algorithm, while both TD3 and SAC perform poorly on one or more tasks, and DDPG fails to train stably on most tasks. The thick lines represent the mean performance for a method at step t across 4 runs, and the upper and lower lines represent the max and min across those runs.

Performance on standard benchmarks When run on 5 different standard benchmark tasks for continuous-valued action space learning (`HalfCheetah`, `Humanoid`, `Hopper`, `Pusher`, and `Ant`), CGP achieves maximum reward competitive with the best methods on that task (with the exception of `Ant`, where TD3 is a clear winner). Importantly, CGP performed consistently well across all tasks, even though its hyperparameters were only optimized on one task- in all tasks it is either the best or second best method. Other methods (i.e. SAC and TD3) perform well on one task with the given set of hyperparameters, such as TD3 on `Ant` or SAC on `Humanoid`, but perform poorly on one or more other tasks, as TD3 performs poorly on `Humanoid` and SAC on `Ant`. I note that for each method better performance can be achieved using hyperparameters tuned for the task (for example, Haarnoja et al. report much better performance on `Humanoid` using task-specific hyperparameters[46]), but as I am interested in inter-task hyperparameter robustness I did not perform such tuning. Additionally, even though CGP is based on the Q-function used in the TD3 method, it greatly outperforms TD3 on complex tasks such as `Humanoid`, suggesting that the CEM policy is more robust across tasks. See Figure 1.2 for details.

Stability across runs Across a wide range of hyperparameters (excluding very large learning rate ≥ 0.01), CGP offers a tight clustering of final evaluation rewards. Other methods demonstrated higher-variance results, where individual runs with slightly different hyperparameters would return significantly different run distributions. To arrive at this conclusion, I ran a large battery of hyperparameter sweeps across methods, the detailed results of which can be observed in Appendix A.1. I consider CGP's relative invariance to hyperparameters that are sub-optimal one of its most valuable attributes; I hope that it can be applied to new problems with little or no hyperparameter tuning.

Robustness across hyperparameters I evaluated the robustness of each method over hyperparameter space, using a common set of hyperparameter configurations (with small differences for specific methods based on the method). For most hyperparameters, I held all others fixed while varying only that pa-

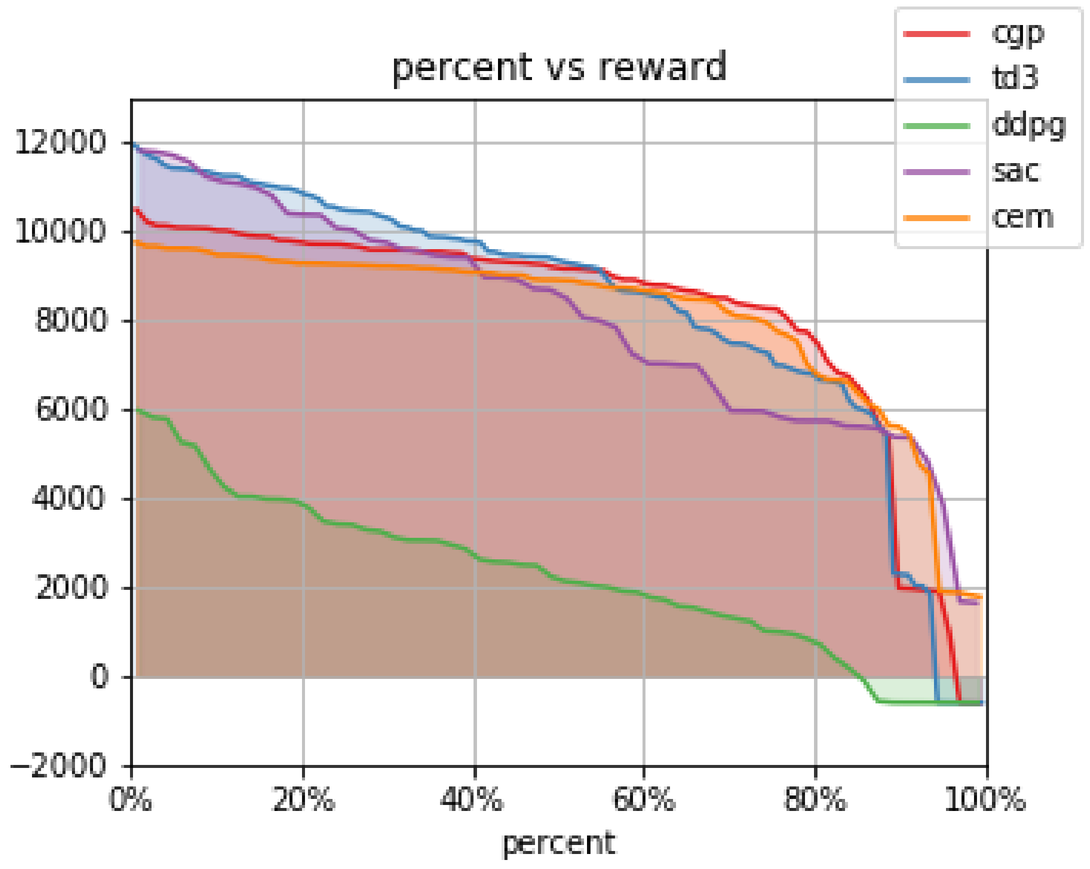


Figure 1.3: Stability of various methods on the `HalfCheetah` benchmark task. This figure shows the percentage of runs across all hyperparameter configurations that reached at least the indicated reward level. While CGP is outperformed with optimized parameters, its performance decays much slower for sub-optimal configurations.

rameter. I varied learning rate (LR) and batch size jointly, with smaller learning rates matching with smaller batch sizes, and vice versa. I varied LR among the set {0.01, 0.001, 0.0001}, and batch size among {256, 128, 64, 32}. I also independently varied the size of the network in {512, 256, 128, 32}. For methods using random sampling for some number of initial timesteps (CGP and TD3), I varied the number in {0, 1000, 10000}, and for those which inject noise (all other than SAC) I varied the exploration and (for CGP and TD3) next action noise in {0.05, 0.1, 0.2, 0.3}. I evaluated CGP entirely with no exploration noise, which other methods using deterministic policies (TD3, DDPG) cannot do while remaining able to learn a non-trivial policy. The overall results of these sweeps can be seen in Figure 1.3, while detailed results breaking the results down by hyperparameter are in the supplement.

Overall, while CGP does not perform as well in the top quartile of parameter sweeps, it displays a high degree of stability over most hyperparameter combinations, and displays better robustness than SAC in the lower half of the range and DDPG everywhere. CGP also performs slightly better for almost all states than the CEM policy it learns to imitate. The failure cases in the tail were, specifically, too high a learning rate (LR of 0.01, which is a failure case for CGP but not for CEM) and less initial random sampling (both 0 and 1000 produced poor policies for some seeds).

Inference speed and training efficiency I next benchmarked the training and inference runtime of each method. I computed the mean over 10 complete training and inference episodes for each method with the same parameters. The results can be found in Table 1.1. CEM-2, CGP-2, CEM-4, and CGP-4 refer to the number of iterations of CEM used(2 or 4). π_{CGP} greatly outperforms π_{CEM} at inference time, and performs the same at training time. Other methods are faster at training time, but run at the same speed at inference time Importantly, the speed of π_{CGP} at inference time is independent of the number of iterations of CEM sampling used for training.

Table 1.1: Runtime in average seconds per episode of `HalfCheetah` (without rendering) on an otherwise-idle machine with a Nvidia GTX 1080 ti GPU. CGP achieves a constant inference runtime independent of the number of CEM iterations used, which matches the performance of other methods.

Method	Mean Train (s)	Mean Inference (s)
Random	-	0.48
DDPG	5.75	2.32
TD3	5.67	2.35
SAC	11.00	2.35
CEM-2	7.1	6.3
CEM-4	9.3	10.1
CGP-2	**11.03**	**2.35**
CGP-4	**14.46**	**2.35**

1.5.3 CGP Variants

I considered several variants of my method, as detailed in Section 1.4.2. I ran each variant on a suite of learning rate and batch size combinations to evaluate their robustness. I tested LR values in {0.001, 0.0001} and batch sizes in {32, 128, 256}. See Figure 1.4 for a comparison of all runs performed.

CGP versus QGP The source of the supervision signal is a critical determinant of the behavior of the policy. Thus it is important to compare the performance of the policy when trained to directly optimize the learned Q-function and when trained to imitate CEM inference on that same policy. I found that directly optimizing the learned Q-function suffers from more instability and sensitivity to chosen hyperparameters, particularly when learning offline. In comparison, both CGP variants train well in most cases. This suggests that the

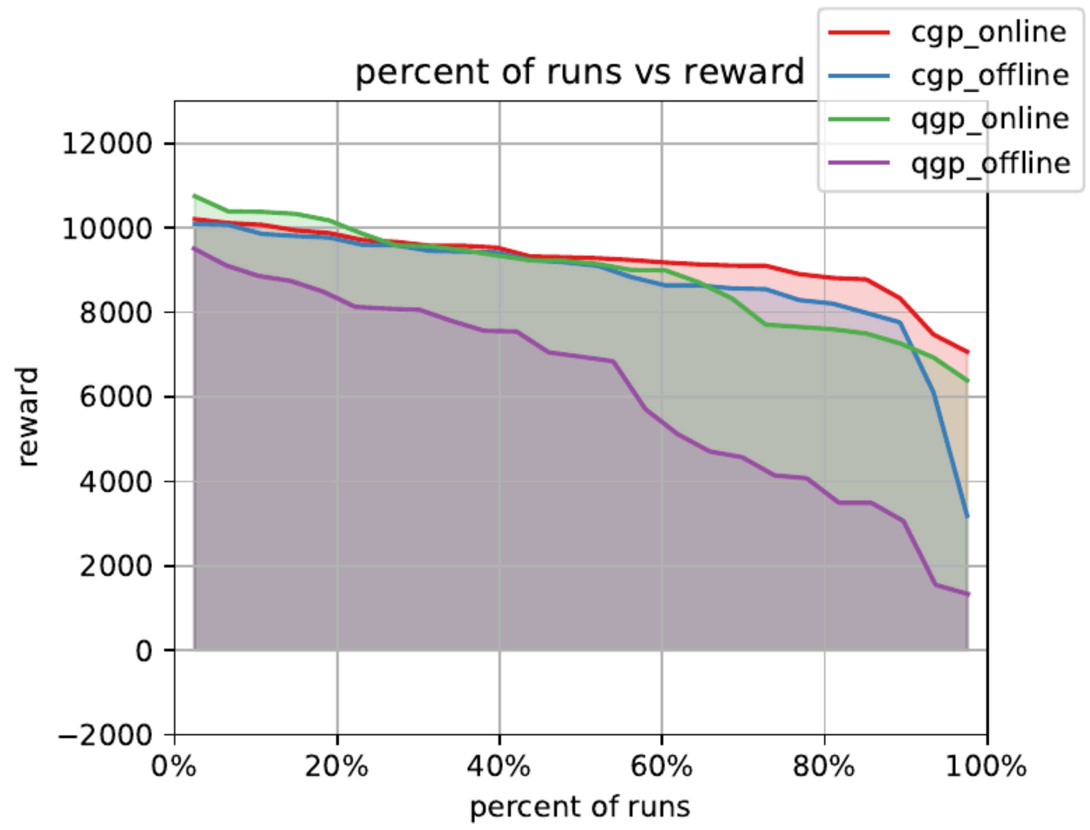


Figure 1.4: Stability of variants of CGP, as measured by the percent of runs across all hyperparameter configurations tested reaching at least the total reward given on the y-axis. Three of the four methods (CGP-Online, CGP-Offline, and QGP-Online) performed roughly equivalently in the upper quartiles, while CGP-Online performed better in the bottom quartile.

CEM approximation to the policy gradient is not only a reasonable approximation but is also easier to optimize than the original gradient.

Online versus Offline Another dimension of customization of CGP is the policy training mode; training can either be online (train the policy function alongside the Q-function, with one policy gradient step per Q-function gradient step) or offline (train only at the end of the Q-function training trajectory). An advantage of the CGP method is that it performs similarly in both paradigms; thus, it is suitable for completely offline training when desired and online learning when the Q-function is available during training.

I found that the online training runs of both CGP and QGP are mildly better than offline training. This result is somewhat intuitive if one considers the implicit regularization provided by learning to optimize a non-stationary Q-function, rather than a static function, as in the offline learning case. Ultimately, CGP is effective in either regime.

1.6 Discussion

In this chapter, I presented Cross-Entropy Guided Policies (CGP) for continuous-action Q-learning. CGP is robust and stable across hyperparameters and random seeds, competitive in performance when compared with state of the art methods, and both faster and more accurate than the underlying CEM policy. Not only is CEM an effective and robust general-purpose optimization method in the context of Q-learning, it is an effective supervision signal for a policy, which can be trained either online or offline. My findings support the conventional wisdom that CEM is a particularly flexible method for reasonably low-dimensional problems[83], and suggest that CEM remains effective even for problems that have potentially high-dimensional latent states, such as Q-learning. I further discuss some future perspectives on this work in Chapter 4.1.

Algorithm 2 CGP: Cross-Entropy Guided Policies

```
TRAINING
Initialize Q-functions $Q_{\theta_1}, Q_{\theta_2}$ and policy $\pi_\phi$ with random parameters $\theta_1, \theta_2, \phi$, respectively
Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
Initialize CEM policies $\pi_{\text{CEM}}^{Q_{\theta_1}}, \pi_{\text{CEM}}^{Q_{\theta'_1}}$
Initialize replay buffer $\mathcal{B}$
Define batch size $b$
for $e = 1$ to $E$ do
    for $t = 1$ to $T$ do
        Step in environment:
        Observe state $s_t$
        Select action $a_t \sim \pi_{\text{CEM}}^{Q_{\theta_1}}(s_t)$
        Observe reward $r_t$, new state $s_{t+1}$
        Save step $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{B}$
        Train on replay buffer ($j \in 1, 2$):
        Sample minibatch $(s_i, a_i, r_i, s_{i+1})$ of size $b$ from $\mathcal{B}$
        Sample actions $\tilde{a}_{i+1} \sim \pi_{\text{CEM}}^{\theta'_1}$
        Compute $q^* = r_i + \gamma \min_{j \in 1,2} Q_{\theta'_j}(s_{i+1}, \tilde{a}_{i+1})$
        Compute losses $\ell_{Q_j} = \left(Q_{\theta_j}(s_i, a_i) - q^*\right)^2$
        CGP loss: $\ell_\pi^{\text{CGP}} = (\pi_\phi(s_i) - \pi_{\text{CEM}}^{\theta_1}(s_i))^2$
        QGP loss: $\ell_\pi^{\text{QGP}} = -Q_{\theta_1}(s_i, \pi_\phi(s_i))$
        Update $\theta_j \leftarrow \theta_j - \eta_Q \nabla_{\theta_j} \ell_{Q_j}$
        Update $\phi \leftarrow \phi - \eta_\pi \nabla_\phi \ell_\pi$
        Update target networks:
        $\theta'_j \leftarrow \tau\theta_j + (1-\tau)\theta'_j, \quad j \in 1, 2$
        $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
    end for
end for
INFERENCE
for $t = 1$ to $T$ do
    Observe state $s_t$
    Select action $a_t \sim \pi_\phi(s_t)$
    Observe reward $r_t$, new state $s_{t+1}$
end for
```

2

Exploration via Reward Prediction Error

2.1 Introduction

In recent years deep reinforcement learning (RL) algorithms have demonstrated impressive performance on tasks such as playing video games and controlling robots[73,55]. However, successful training for such cases typically requires both a well-shaped reward function, where the RL agent can sample improved trajectories through simple dithering exploration such as ϵ-greedy sampling, and the ability to collect many (hundreds of thousands to millions) of trials. Satisfying these preconditions often requires large amounts of domain-specific engineering. In particular, reward function design can be unintuitive, may require many iterations of design, and in some domains such as robotics can be physically impractical to implement.

The field of *exploration* methods in RL seeks to address the difficulties of reward design by allowing RL agents to learn from *unshaped* reward functions. Unshaped functions (for example, a reward of 1 when an object is moved to a target, and 0 otherwise) are usually much easier to design and implement than dense well-shaped reward functions. However, it is hard for standard RL algorithms to discover good policies on unshaped reward functions, and they may learn very slowly, if at all.

While substantial work has been conducted on designing general exploration strategies for high-dimensional Markov Decision Processes (MDPs) with sparse reward functions, few studies have distinguished between different types of tasks requiring exploration, particularly in terms of which signals in each MDP are useful for discovering new sources of reward. In this work, I consider three types of exploration challenges in particular: solving mazes, learning goal conditioning relationships, and escaping local reward maxima.

Many classical exploration tasks can be described well as **mazes**. For example, discovering the single rewarding state in a sparse reward environment, or navigating a precise series of obstacles in order to play a game. Qualitatively, the agent must search for the exit to the maze (reward), receives little or no reward before finding it, and has no learned priors. In the limit any RL task can be seen as a maze (such as by treating a single optimal trajectory as the "exit"), but such a treatment is often intractable for large MDPs.

Related but distinct are **goal conditioned tasks**. Here, the reward function is conditioned on a non-static goal specified by the environment and discovered through interaction. For example, a robot that must move an object to a set of coordinates, which differ for each episode. The agent must learn how the observation and reward are conditioned on the goal, which is made significantly harder when the underlying reward function is sparse and unshaped. Unlike a maze, correlations between observation and goal/reward provide additional information an agent can use to solve the problem.

Lastly, in a poorly-shaped reward function there may exist local maxima in the space of trajectories, where an agent cannot discover an improved policy through local exploration and must deliberately sample suboptimal trajectories to **escape local maxima**. Here, the contours of the reward function can provide information on what directions of exploration might be informative, even if exploring them does not immediately maximize reward.

This distinction is important because both goal conditioning and local maxima introduce additional information about the task that mazes do not contain — In a maze-like environment, discovering new states is explicitly linked with discovering new reward signals. Goal conditioning can provide hints as to what states are and are not rewarding (and when) through correlations in the observation. Similarly, local reward maxima are embedded within a dense reward function which provides correlations between each observation and the reward that results, and discovering this relationship may lead to improved reward. Each of these problems can be intractable using naive exploration (depending on the severity of the problem), but each in turn provides some signal that can be used to solve it. For example, goal-conditioning relationships can also be learned by goal-driven RL methods such as Hindsight Experience Replay[6], which by assuming the presence of a goal can learn much faster and more sample-efficiently on that class of problems.

In this paper, I propose the use of reward prediction error, specifically the Temporal-Difference Error (TD-Error) of a value function, to direct exploration in MDPs that contain Goal Conditioning and Local Maxima Escape problems but do not have a strong correlation between reward discovery and state novelty. To facilitate the use of this objective in a deep reinforcement learning setting for high-dimensional MDPs, I introduce QXplore, a new deep RL exploration formulation that seeks novelty in the predicted reward landscape instead of novelty in the state space. QXplore exploits the inherent reward-space signal from TD-error in value-based RL, and directly promotes visiting states where the current understanding of reward dynamics is poor. In the following sections, I describe QXplore for continuous MDPs and demonstrate its utility for efficient learning on a variety of complex benchmark environments showcasing different exploration cases.

2.2 Related Work

Of the exploration methods proposed for deep RL settings, the majority provide some state-novelty objective that incentivizes an agent to explore novel states or transition dynamics. A simple approach consists of explicitly counting how many times each state has been visited, and acting to visit rarely explored states. This approach can be useful for small MDPs, but often performs poorly in high-dimensional or continuous state spaces. However, several recent works[106,8,32] using count-like statistics have shown success on benchmark tasks with complex state spaces.

Another approach to environment novelty learns a model of the environment's transition dynamics and considers novelty as the error of the model in predicting future states or transitions. This exploration method relies on the assumption that any new state that can be predicted in advance is equivalent to some previously seen state in its effect on reward. Predictions of the transition dynamics can be directly computed[78,101], or related to an information gain objective on the state space, as described in VIME[53] and EMI[60].

Several exploration methods have recently been proposed that capitalize on the function approximation properties of neural network to recognize novel states. Random network distillation (RND) trains a function to predict the output of a randomly-initialized neural network from an input state, and uses the approximation error as a reward bonus for a separately-trained RL agent[14]. Similarly, DORA[30] trains a network to predict zero on observed states and deviations from zero are used to indicate unexplored states.

These methods have been shown to perform well on maze-solving exploration tasks such as the Atari game `Montezuma's Revenge`, where maximizing reward (game score) requires visiting each room of the game, which also maximizes the diversity of states and observations experienced. However, evaluating these methods on tasks where novelty does not correlate highly with reward, such as on other Atari games, shows little improvement over ϵ-greedy[105].

Reward prediction error has been previously used for exploration in a few cases. Previous works described using reward misprediction and model prediction error for exploration[89,108]. However, these works were primarily concerned with model-building and system-identification in small MDPs, and used single-step reward prediction error rather than TD-error. Later, TD-error was used as a negative signal to constrain exploration to focus on states that are well understood by the value function for safe exploration[37]. Related to maximizing TD-error is maximizing the variance or KL-divergence of a posterior distribution over MDPs or Q-functions, which can be used as a measure of uncertainty about rewards[30,76]. Posterior uncertainty over Q-functions can be used for information gain in the reward or Q-function space, but posterior uncertainty methods have thus-far largely been used for local exploration as an alternative to dithering methods such as ϵ-greedy sampling, though Osband et al. do apply posterior uncertainty to `Montezuma's Revenge` and other exploration tasks in the Atari game benchmark.

2.3 Preliminaries

I use in this work the RL terminology of Sutton & Barto, in which an agent seeks to maximize reward in a Markov Decision Process (MDP). An MDP consists of states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, a state transition function $S : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ giving the probability of moving to state s_{t+1} after taking action a_t from state s_t for discrete timesteps $t \in 0, ..., T$. Rewards are sampled from reward function $r : \mathcal{S} \times \mathcal{A} \to R$. An RL agent has a policy $\pi(s_t, a_t) = p(a_t|s_t)$ that gives the probability of taking action a_t when in state s_t. The agent aims to learn a policy to maximize the expectation of the time-decayed sum of reward $R_\pi(s_0) = \sum_{t=0}^{T} \gamma^t r(s_t, a_t)$ where $a_t \sim \pi(s_t, a_t)$.

A value function $V_\theta(s_t)$ with parameters θ is a function which computes $V_\theta(s_t) \approx R_\pi(s_t)$ for some policy π. Temporal difference (TD) error δ_t measures the bootstrapped error between the value function at the current timestep and the next timestep as

$$\delta_t = V_\theta(s_t) - (r(s_t, a_t \sim \pi(s_t)) + \gamma V_\theta(s_{t+1})). \tag{2.1}$$

A Q-function is a value function of the form $Q(s_t, a_t)$, which computes the expected future reward $Q(s_t, a_t) = r(s_t, a_t) + \gamma \cdot \max_{a'} Q(s_{t+1}, a')$, assuming the optimal action is taken at each future timestep. An approximation to this optimal Q-function Q_θ with some parameters θ may be trained using a mean squared TD-error objective $L_{Q_\theta} = ||Q_\theta(s_t, a_t) - (r(s_t, a_t) + \gamma \cdot \max_{a'} Q'_{\theta'}(s_{t+1}, a'))||^2$ given some target Q-function $Q'_{\theta'}$, commonly a time-delayed version of Q_θ[73]. Extracting a policy π given Q_θ amounts to computing $\text{argmax}_a Q_\theta(s_t, a)$.

2.4 QXplore: TD-Error as Reward Signal

2.4.1 TD-error Objective

I first discuss why and how TD-error can be used as an exploration signal in deep RL settings on the classes of MDPs discussed above. Many Deep RL methods maintain a value function, typically a Q function, which in off-policy settings is bootstrapped to approximate the true Q function of the optimal policy. During the course of training, this Q function will naturally contain inaccuracies such that there is nontrivial Bellman error for certain (s, a, s', r) tuples. Intuitively, these errors indicate that the current estimate of the Q function does not correctly model the reward dynamics of the MDP per Bellman optimality. Therefore, an exploration method that prioritizes seeking out regions of the environment where the Q-function is inaccurate could aid an off-policy method in discovering novel sources of reward and propagating those improvements through the Q function.

Given a Q function with parameters θ and δ_t I define the exploration signal for a given state-action-next-state tuple as:

$$\begin{aligned} r_{x,\theta}(s_t, a_t, s_{t+1}) = |\delta_t| = |Q_\theta(s_t, a_t) - \\ (r_{\mathrm{E}}(s_t, a_t) + \gamma \max_{a'} Q'_{\theta'}(s_{t+1}, a'))| \end{aligned} \tag{2.2}$$

for some extrinsic reward function r_{E} and target Q-function Q'_θ. Notably, I use the absolute value of the TD rather than signed TD, as this is necessary to harness network extrapolation error in sparse reward environments.

Intuitively, a policy maximizing the expected sum of r_x for a fixed Q function will sample trajectories where Q_θ does not have an accurate estimate of the future rewards it will experience. This is useful for exploration because r_x will be large not only for state-action pairs producing unexpected reward, but for all state-action pairs leading to such states, providing a denser exploration reward function and allowing for longer-range exploration.

2.4.2 Q_x: Learning a Q-Function to Maximize TD-error

Now that we have defined a TD-error exploration formulation, we must ask, how should we maximize it? If we treat this signal as a reward function, r_x can be used to generate a new MDP where the reward function is replaced by r_x, and thus generally can be solved via any RL algorithm. In practical terms, we can choose to train a second Q-function to maximize r_x, which allows the entire algorithm to be trained off-policy and for the two Q-functions to share replay data. Additionally, this allows us to use the original Q function Q_θ as an exploitation policy at inference time, avoiding the need to trade off between exploration and exploitation because the Q function estimates are not directly affected by r_x values.

In this formulation, which I call QXplore, I define a Q-function, $Q_{x,\phi}(s, a)$ with parameters ϕ, whose reward objective is r_x. I train $Q_{x,\phi}$ using the standard bootstrapped loss function

$$L_{Q_{x,\phi}} = ||Q_{x,\phi}(s_t, a_t) - (r_x(s_t, a_t, s_{t+1})+ \\ \gamma \max_{a'} Q'_{x,\phi'}(s_{t+1}, a'))||^2. \quad (2.3)$$

The two Q-functions, Q_θ and Q_x, are trained in parallel, sharing replay data so that Q_θ can learn to exploit sources of reward discovered by Q_x and so that Q_x can better predict the TD-errors of Q_θ. Since the two share data, π_{Q_x} acts as an adversarial teacher for Q_θ, sampling trajectories that produce high TD-error under Q_θ and thus provide novel information about the reward landscape. A similar adversarial sampling scheme was used to train an inverse dynamics model by Hong et al., and Colas et al. use separate goal-driven exploration and reward maximization phases for efficient learning. However, to my knowledge adversarial sampling policies have not previously been used for exploration. To avoid off-policy stability issues due to the different reward objectives, I sampled a fixed ratio of experiences collected by each policy for each training batch. The full method is described for the continuous-action domain in Algorithm 3 and a schematic of the method is shown in Figure 2.1.

2.4.3 State Novelty from Neural Network Approximation Error

A key question in using TD-error for exploration is what happens when the reward landscape is flat? Theoretically, in the case that $\forall (s, a), r(s, a) = C$ for some constant $C \in R$, an optimal Q-function which generalizes perfectly to

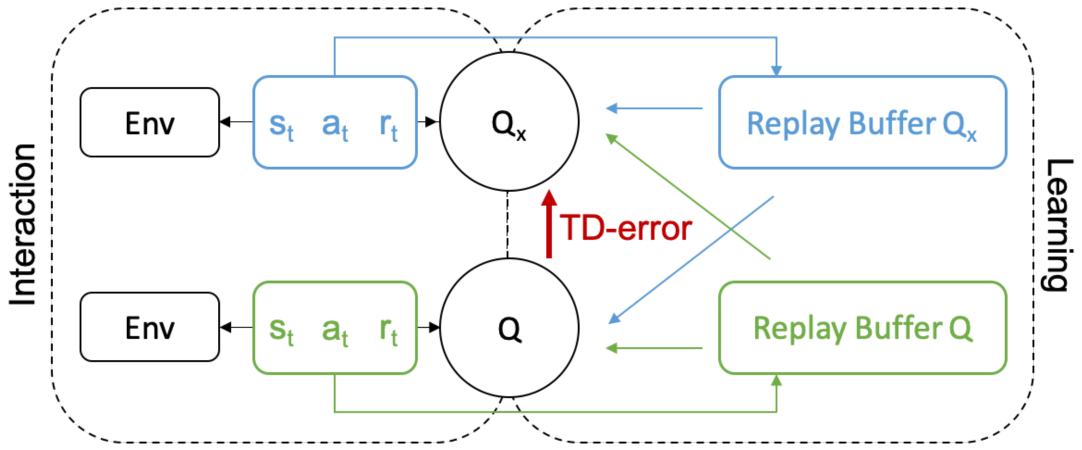


Figure 2.1: Method diagram for QXplore. The method uses two Q-functions which sample trajectories from their environment and store experiences in separate replay buffers. Q is a standard state-action value-function, whereas Q_x's reward function is the unsigned temporal difference error of the current Q on data sampled from both replay buffers. A policy defined by Q_x samples experiences that maximize the TD-error of Q, while a policy defined by Q samples experiences that maximize discounted reward from the environment.

unseen states will simply output $\forall(s,a), Q^\star(s,a) = \sum_{t=0}^{\infty} C\gamma^t$ in the infinite time horizon case. This results in a TD-error of 0 everywhere and thus no exploration signal. However, using neural network function approximation, I found that perfect generalization to unseen states-action pairs does not occur, and in fact observe in Figure 2.2 that the distance of a new datum from the training data manifold correlates with the magnitude of the network output's deviation from $\sum_{t=1}^{\infty} C\gamma^t$ and thus with TD-error. As a result, in the case where the reward landscape is flat TD-error exploration converges to a form of state novelty exploration. This property of neural network function approximation has been used by several previous exploration methods to good effect, including RND[14] and DORA[30]. In particular, the exploration signal used by RND (extrapolation

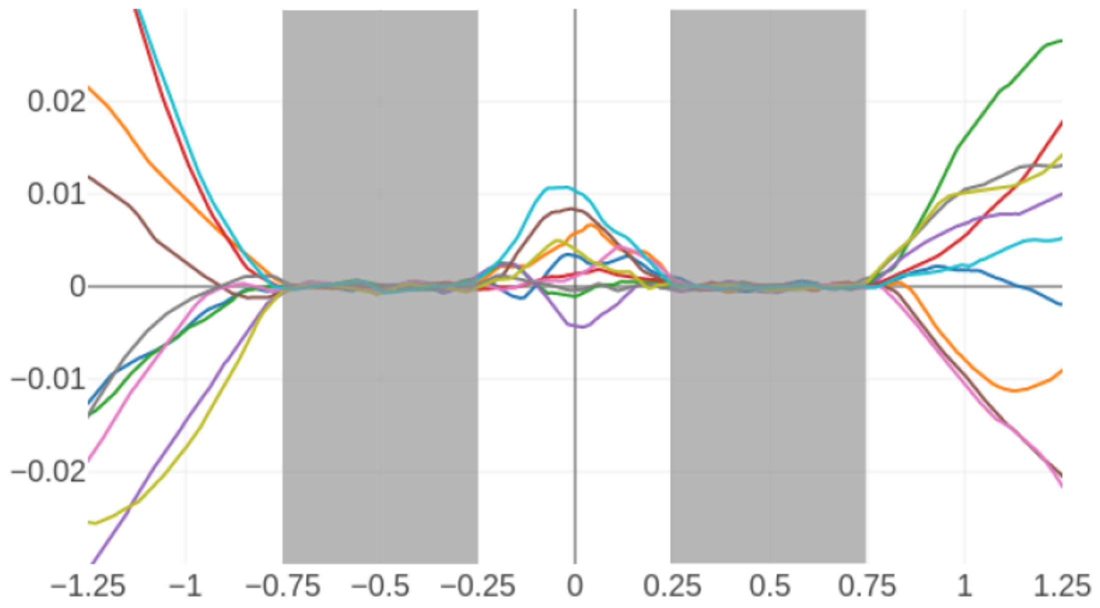


Figure 2.2: A neural network trained to predict a constant value does not interpolate or extrapolate well outside its training range, which can be exploited for exploration. Predictions of 3-layer MLPs of 256 hidden units per layer trained to imitate $f(x) = 0$ on $R \rightarrow R$ with training data sampled uniformly from the shaded regions. Each line is the final response curve of an independently trained network once its training error has converged.

error from fitting the output of a random network) should be analogous to r_x (extrapolation error from fitting a constant value), meaning the QXplore should perform comparably to RND when no extrinsic reward exists.

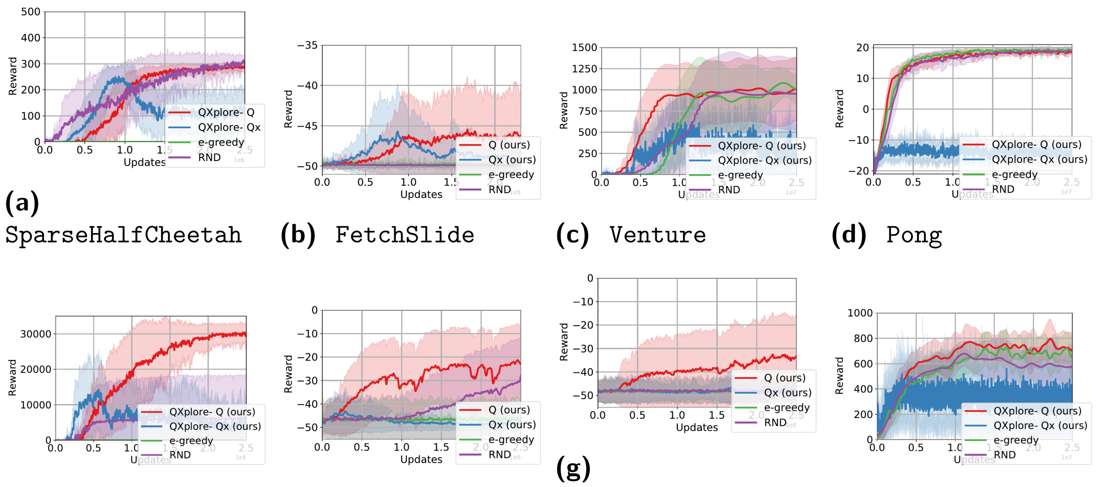
2.5 Experiments

I describe in this section the results of experiments to demonstrate the effectiveness of QXplore on continuous control and Atari benchmark tasks. I also compare to results on `SparseHalfCheetah` from several previous publications. Finally, I discuss several ablations to QXplore to demonstrate that all components of the method improve performance.

I compared QXplore primarily with a related state of the art state novelty-based method, RND[14], and with ϵ-greedy sampling as a simple baseline. Each method is implemented in a shared code base on top of TD3/dueling double deep Q-networks for the continuous/discrete action case[33,112]. For experiments in continuous control environments, I implemented a nonparametric cross-entropy method policy, previously described as more robust to hyperparameter variance, with the same architecture and hyperparameters as prior work[98,55]. I experimented with a variant using DDPG-style parametric policies[66] for both Q_θ and Q_x, but found preventing Q_θ's policy from converging to poor local maxima difficult, consistent with previously reported stability issues in that class of algorithms[98,54]. For all experiments, I set the data sampling ratios of Q_θ and Q_x, $\mathcal{R}_Q$ and $\mathcal{R}_{Q_x}$ respectively, at 0.75, the best ratio among a sweep of ratios 0.0, 0.25, 0.5, and 0.75 on `SparseHalfCheetah`. For continuous control tasks, I used a learning rate of 0.0001 for both Q-functions, the best among all paired combinations of 0.01, 0.001, and 0.0001, and fully-connected networks of two hidden layers of 256 neurons to represent each Q-function, with no shared parameters. For Atari benchmark tasks, I used the dueling double deep Q-networks architecture and hyperparameters described by Wang et al.. A full table of hyperparameters for all methods is provided in Appendix B.1.

2.5.1 Experimental Setup

I evaluated QXplore on five continuous control tasks using MuJoCo that require exploration due to sparse or unshaped rewards. First, the `SparseHalfCheetah` task originally proposed by VIME[53]. This task requires an agent to move 5 units (several hundred timesteps of actions) forward to receive reward, receiving 0 reward otherwise, and is maze-like in this regard. Next, I evaluated on three goal-directed OpenAI gym tasks, `FetchPush`, `FetchSlide` and `FetchPickAndPlace`, originally proposed in HER[6]. Lastly, I tested a variant of `SparseHalfCheetah` that I call `LocalMaxEscape` where a local reward maximum has been introduced — the agent receives 0 reward for every timestep it is between -1 and 1 units from the origin, -1 reward if it moves outside that range, but 100 reward per timestep if it moves 5 units forward, similar to `SparseHalfCheetah`. I chose these tasks as they are challenging exploration problems highlighting the three cases of interest that are relatively simple to control, but still involve large continuous state spaces and continuous actions. Guided by a recent study suggesting that exploration in the Atari game benchmark suite doesn't improve performance on most tasks[105], I evaluated on a pair of "hard" exploration games, `Venture` and `Gravitar`, as well as an easy game, `Pong`, to show that QXplore can also function in this very different domain. I ran five random seeds for each experiment and plot the mean and plus/minus 1 standard deviation bounds for each set of runs, applying a Gaussian filter to each mean/stdev for readability.



(a) SparseHalfCheetah (b) FetchSlide (c) Venture (d) Pong

(e) LocalMaxEscape (f) FetchPush (g) FetchPickAndPlace (h) Gravitar

Figure 2.3: Performance of QXplore compared with RND and ϵ-greedy sampling. QXplore outperforms RND and ϵ-greedy on the Fetch tasks and in escaping local maxima, while performing comparably on maze solving tasks and non-exploration tasks. "QXplore-Q" indicates the performance of the exploitation Q-function, while "QXplore-Qx" indicates the performance of the exploration Q-function, whose objective does not directly maximize reward but which may lead to high reward regardless.

2.5.2 Exploration Benchmark Performance

I show the performance of each method on each task in Figure 2.3. QXplore performs comparablly to RND on the `SparseHalfCheetah` task, in line with the intuition from Section 2.4.3, but performs much better comparatively on the `Fetch` tasks — only on `FetchPush`, the easiest task, did RND find non-random reward. I believe this is because TD-error drives exploration behavior that helps the agent to uncover the goal-conditioning relationship, whereas state novelty is goal-agnostic and does not aid in discovery of the relationship. QXplore also strongly outperformed RND on `LocalMaxEscape`, as negative rewards far from the origin increase TD error and drive rapid discovery of the global optimum.

Episodes until mean reward of	**QXplore**	**VIME**	**EX2**	**EMI**
50	3000	10000*	4740*	**2580***
100	**3400**	x*	6180*	4520*
200	**4000**	x*	x*	8440*
300	**10000**	x*	x*	x*

Table 2.1: Number of episodes required to reach mean reward milestones on `SparseHalfCheetah` for several methods. QXplore reaches higher rewards than previously published results. Results marked with "*" are previously published numbers. Results marked with "x" indicate that the mean reward was not achieved.

To validate performance and sample efficiency, I compare QXplore to previously published `SparseHalfCheetah` performance numbers in Table 2.1. Because to my knowledge no previous work has evaluated off-policy Q-learning based methods on `SparseHalfCheetah`, I compare to previous methods built on top of TRPO[90]. Due to the difference in baseline algorithms, I compare the number of episodes of interaction required to reach a given level of reward, though QXplore was not intended to be performant with respect to this metric. While some decrease in sample efficiency is expected due to differing baseline methods (TRPO[90] versus TD3[33]), compared to the results reported by Kim et al. for EMI[60] and EX2[32], and by Houthooft et al. for VIME[53] on the `SparseHalfCheetah` task, QXplore reaches almost every reward milestone faster, and achieves a peak reward (300) not achieved by any previous method. This shows that off-policy Q-learning combined with TD-error exploration can result in sample efficient as well as flexible exploration.

I also implemented a continuous-control adaptation of DORA[30] and tested it on `SparseHalfCheetah`. DORA performed poorly, possibly because it was not intended for use with continuous action spaces, and thus I did not test it on other tasks.

As a comparison to a published off-policy Q-learning exploration method, I considered GEP-PG[18], which used separate exploration and exploitation phases similar to QXplore. I downloaded the author's implementation (built on top of DDPG) and tested it on `SparseHalfCheetah` using the parameters for the `HalfCheetah` task it was originally tested on. GEP-PG reached a validation reward of 120.2 after 4000 episodes, broadly comparable to my QXplore and RND implementations.

Finally, while the main focus of this evaluation is on continuous control tasks, I also evaluated QXplore on several games in the Atari Arcade Learning Environment[9] to verify that QXplore extends to tasks with image observations and discrete action spaces. I implemented QXplore and RND on top of dueling double DQN[112], using hyperparameters and network architectures from the Dopamine implementation of DQN[16]. The results are shown in Figure 2.3 after 25 million training steps. Based on the findings of Taiga et al., I did not expect to improve significantly compared to the baseline in this domain. Indeed, I find that QXplore performs comparably to the baseline and RND implementations on `Pong` and `Gravitar`, while outperforming them modestly on `Venture`, where QXplore converges faster, perhaps due to Q_x focusing on reward-adjacent states more than ϵ-greedy or RND.

2.5.3 Ablations

There are two major features of QXplore that distinguish it from prior work in exploration: the use of a pair of policies that share replay data, and the use of unsigned TD-error to drive exploration. I performed several ablations that assess the contribution of each of these features to the method and confirm their value for exploration. The results are shown in Figure 2.4. I found that the use of separate exploration and exploitation policies along with unsigned TD-error is necessary to obtain good performance, and that ablations of these components either fail to train or substantially reduce performance. I discuss each case in detail below.

Single-Policy QXplore. First, I tested a single-policy version of QXplore by replacing $Q_\theta(s, a)$ with a value function $V_\theta(s)$. I used a value function rather than Q-function in this case to avoid large estimation errors stemming from fully off-policy training. I observe in Figure 2.4 that while the policy is able to find reward quickly and converge faster, the need to satisfy both objectives results in a lower converged reward than the original QXplore method.

1-Step Reward Prediction. Second, I ran an ablation where I replace $Q_\theta(s, a)$ with a function that simply predicts the current $r(s_t, a_t)$. Using reward error instead of a value function in Q_x can still produce the same state novelty fallback behavior in the absence of reward; however, it provides only limited reward-based exploration utility. I evaluated this variant and observe in Figure

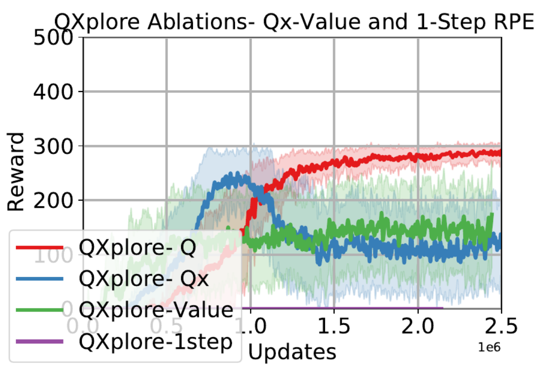
2.4 that it fails to sample reward. Reward prediction error is not sufficient to allow strong exploration behavior without some form of look-ahead mechanism.

QXplore with State Novelty Exploration. To assess the importance of TD-error specifically in the algorithm, I replaced the TD-error maximization objective of Q_x with the random network prediction error maximization objective of RND, while still performing rollouts of both policies. The results are shown in Figure 2.4. While the modified Q_x samples reward, it is too infrequent to guide Q to learn the task. Qualitatively, the modified Q_x function does not display the directional preference in exploration that normal Q_x does once reward is discovered, instead sampling both directions equally.

QXplore with Signed TD-Error Objective. While I used unsigned TD-error to train Q_x, I also tested QXplore using signed TD-error. I used the negative signed TD-error $-\delta_t$ from equation 2.1 so that better-than-expected rewards result in positive r_x values. The results of this experiment are shown in Figure 2.4. While this ablation is able to converge and solve the task, the unsigned TD-error performs much better on `SparseHalfCheetah`, likely due to the extrapolation error described in Figure 2.2 being both positive and negative.

2.5.4 Qualitative Behavioral Analysis

Qualitatively, on `SparseHalfCheetah` I observed some interesting behavior from Q_x late in training. After initially converging to obtain high reward, Q_x appears to get “bored” and will focus on the reward threshold, stopping short or jump-



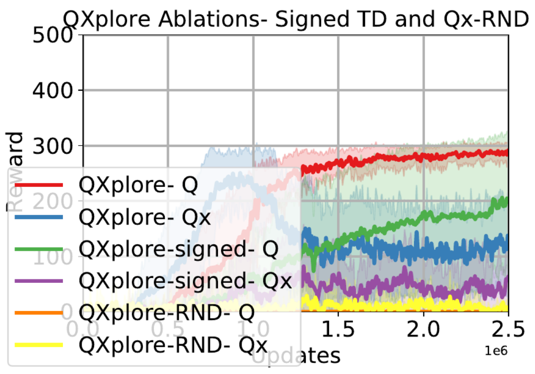


Figure 2.4: Plots showing several ablations of QXplore on `SparseHalfCheetah`. While several variants are able to learn the task, the full QXplore formulation performs better.

ing back and forth across it, which results in reduced reward but higher TD-error. This behavior is distinctive of TD-error seeking over state novelty seeking, as such states are not novel compared to moving past the threshold but do result in higher TD-error. Such behavior from Q_x motivates Q to sample the state space around the reward boundary and thus learn to solve the task. Example sequences of such behaviors are shown in Figure 2.5.

2.6 Discussion

Here, I proposed the use of reward prediction error as an objective for exploration in deep reinforcement learning. I defined a deep RL algorithm, QXplore, using TD-error that is sufficient to discover solutions to multiple types of challenging exploration tasks across multiple domains. I found that QXplore performs well across all exploration task types tested compared to my state novelty baseline, although type-specific algorithms can likely perform better on some

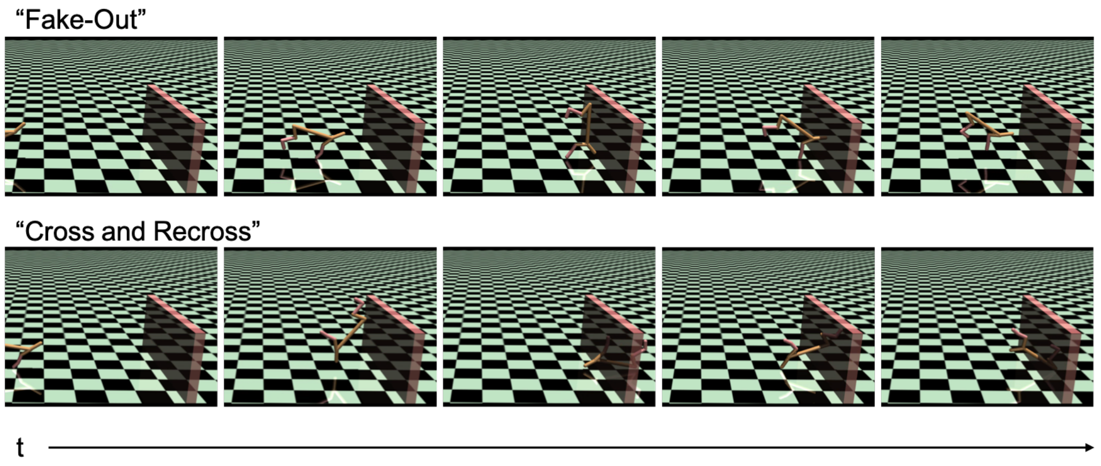


Figure 2.5: Example trajectories showing Q_x's behavior late in training that is distinctive of TD-error maximization. The corresponding Q network reliably achieves reward at this point. In “fake-out”, Q_x approaches the reward threshold and suddenly stops itself. In “cross and recross”, Q_x crosses the reward threshold going forward and then goes backwards through the threshold.

types, such as goal-directed exploration.

While QXplore is a general-purpose exploration algorithm that can be applied successfully to many tasks, several limitations remain for TD-error exploration. In the worst-case, TD-error likely performs no better than state novelty for certain “pure” exploration tasks, such as exploring a linear chain of states, though with an optimistic prior on the Q-values of unseen states it may perform comparably to state novelty. There also exist adversarial tasks where the unsigned TD-error leads to less efficient exploration compared to other possible policies, such as a task with many states that yield large negative rewards uncorrelated with positive rewards. Combining TD-error exploration with reward exploitation may help in such cases to bias the search. However, balancing the rate at which the TD-error signal disappears for a given state with the reward function's magnitude is critical to get rapid convergence, and more research into such approaches is needed. Lastly, TD-error maximization may result in "risky" exploration (in contrast to the "safe" TD-minimizing exploration of Gehring & Precup) and thus may not be well suited for tasks where failures or negative returns have real-world consequences without additional constraints on the agent's actions, or the use of signed TD-error to avoid trajectories yielding worse-than-expected returns.

I hope that these results can spur more investigation into TD-error-based exploration methods to address some of the outstanding challenges described above, as well as encourage further work on diverse exploration signals in RL and on more general exploration objectives suitable for use on heterogeneous RL tasks. I will discuss some such future directions in Chapter 4.2.

Algorithm 3 QXplore for Continuous Actions

```
Input: MDP $S$, Q-function $Q_\theta$ with target $Q'_{\theta'}$, $Q_x$ function $Q_{x,\phi}$ with target $Q'_{x,\phi'}$, replay buffers $\mathcal{Z}_Q$ and $\mathcal{Z}_{Q_x}$, batch size $B$ and sampling ratios $\mathcal{R}_Q$ and $\mathcal{R}_{Q_x}$, CEM policies $\pi_Q$ and $\pi_{Q_x}$, time decay parameter $\gamma$, soft target update rate $\tau$, and environments $E_Q$, $E_{Q_x}$
while not converged do
    Reset $E_Q$, $E_{Q_x}$
    while $E_Q$ and $E_{Q_x}$ are not done do
        Sample environments
        $\mathcal{Z}_Q \leftarrow (s, a, r, s') \sim \pi_Q | E_Q$
        $\mathcal{Z}_{Q_x} \leftarrow (s, a, r, s') \sim \pi_{Q_x} | E_{Q_x}$
        Sample minibatches for $Q_\theta$ and $Q_{x,\phi}$
        $(s_Q, a_Q, r_Q, s'_Q) \leftarrow B{*}\mathcal{R}_Q$ samples from $\mathcal{Z}_Q$ and $B{*}(1{-}\mathcal{R}_Q)$ samples from $\mathcal{Z}_{Q_x}$
        $(s_{Q_x}, a_{Q_x}, r_{Q_x}, s'_{Q_x}) \leftarrow B * \mathcal{R}_{Q_x}$ samples from $\mathcal{Z}_{Q_x}$ and $B * (1 - \mathcal{R}_{Q_x})$ samples from $\mathcal{Z}_Q$
        Train
        $r_{x,\theta} \leftarrow |Q_\theta(s_{Q_x}, a_{Q_x}) - (r_{Q_x} + \gamma Q'_{\theta'}(s'_{Q_x}, \pi_Q(s'_{Q_x})))|$
        $L_Q \leftarrow ||Q_\theta(s_Q, a_Q) - (r_Q + \gamma Q'_{\theta'}(s'_Q, \pi_Q(s'_Q)))||^2$
        $L_{Q_x} \leftarrow ||Q_{x,\phi}(s_{Q_x}, a_{Q_x}) - (r_{x,\theta} + \gamma Q'_{x,\phi'}(s'_{Q_x}, \pi_{Q_x}(s'_{Q_x})))||^2$
        Update $\theta \propto L_Q$
        Update $\phi \propto L_{Q_x}$
        $\theta' \leftarrow (1 - \tau)\theta' + \tau\theta$
        $\phi' \leftarrow (1 - \tau)\phi' + \tau\phi$
    end while
end while
```

3

Q-Function Prioritized Tree Search

3.1 Introduction

Within both the programming languages and machine learning communities, there has been a renaissance in *program synthesis*, the task of automatically generating computer code from a user-provided specification. Due to increased computational power and the rise of deep neural networks, this once intractable problem has now become realizable, and has already seen use in many domains[40], including improving programmer efficiency by automating routine tasks[27] providing an intuitive interface for non-experts without programming knowledge[39], and reducing bugs and improving runtime efficiency for performance-critical code[88].

Within the programming languages (PL) community, program synthesis is typically solved using *enumerative search* – finding correct programs for a given specification by naively enumerating candidates until a satisfying program is found[28,31,3]. This approach is made tractable by narrowing the search space through integrating deductive components into the search process[57,61,63], or by modifying the language of interest into an equivalent language with a narrower search space, and searching within that space[71,41,79]. Another common approach is to reduce synthesis tasks to finding a satisfying assignment to a Boolean formula via a SAT solver[100,5], but this merely pushes the enumerative search into the solver.

Researchers in the machine learning(ML) community have attacked this problem from a different direction – rather than searching naively through a restricted space, correct programs can be efficiently found within a larger search space by intelligently searching or sampling from that space using a learned model of how specifications map to programs. Most of these methods use a supervised learning approach – training on a large synthetic dataset of input/output examples and corresponding satisfying programs, then using recurrent neural network

(RNN)-based models to output each line in the target program successively given a corresponding set of input/output examples[21,99,7,13,36]. Conceptually, this approach should compose well with PL techniques – intelligently searching through a small search space is much easier than intelligently searching through a large search space. Why then are these techniques not commonly used together?

Most existing work in the ML community relies heavily on the structure of the domain specific language (DSL) being synthesized to achieve good performance, and can't generalize to new languages easily. These methods often require language features such as full differentiability of the language[13,36] or a large training set of specification and satisfying program pairs[21,99,7]. These requirements make it difficult to combine existing ML-based methods with techniques developed by the PL community, which require very different properties in a DSL for effective synthesis. Further, most existing methods have heavily focused on solving programs in a "one-shot" fashion, either successfully outputting a satisfying program for a given specification in a single or small number of attempts or failing to do so, which scales poorly as program spaces become larger. In contrast, as search-based methods enumerate all programs, they can solve any synthesis task eventually, but they may take infeasible amounts of time to do so.

Reinforcement Learning Guided Tree Search

My method, *reinforcement learning guided tree search (RLGTS)*, illustrated in figure 3.1, seeks to combine the benefits of both search-based and ML-based

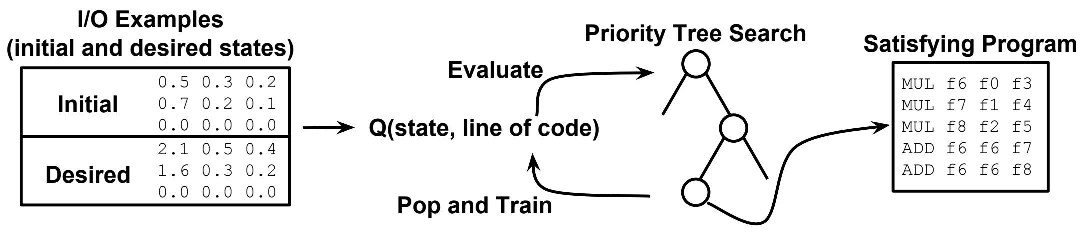


Figure 3.1: Schematic view of RLGTS. This approach to program synthesis treats the problem as a Markov decision process solvable by reinforcement learning, and combines this with a prioritized search tree to speed up the solving process by avoiding local minima, improving the number of programs solvable within a fixed number of attempts. Given a set of input memory states and corresponding output memory states, RLGTS seeks to learn a policy which outputs a sequence of lines of code that maps each input example to the corresponding output example, using a reward function defined for partially-correct solutions to guide the learning process.

methods, and allow for the combination of techniques from both research communities to further enhance performance. I propose a new approach for program synthesis, representing the process of synthesizing a program as a Markov Decision Process(MDP)[102] and using reinforcement learning(RL) to learn to solve a program given only a set of input/output examples for that program, a language specification, and a reward function for the *quality* of a given program. In my RL-based approach, I interpret the program state and current partial program as an environment, and lines of code in the language as actions seeking to maximize the reward function. Furthermore, I combine my RL model with a tree-based search technique which dramatically improves the performance of the method. This combination helps address issues of local minima and efficient sampling which arise in many RL applications.

RLGTS does not depend on the availability of training data for a given language, and makes no assumptions about the structure of the language other than that the language allows for partial programs to be executed and evaluated. Further, my RL-based approach can be combined with other program synthesis methods easily and naturally, allowing for users to benefit from the extensive work on search-based synthesis available for some domains.

In summary, in this chapter I do the following:

1. I introduce reinforcement learning guided tree search, an approach to program synthesis that interprets program generation as a reinforcement learning task.

2. I describe an implementation of RLGTS on a subset of the RISC-V assembly language, and create an RL agent for this task by combining a Q-network-based policy with a simple search tree method.

3. I demonstrate improvements in the fraction of programs solved of up to 100% and 800% compared to RL-only and enumerative search-only baselines respectively on a synthetic dataset of random programs. Further, I compare RLGTS to a Markov chain Monte Carlo(MCMC) based method that has been used to great success in super-optimizing x86 code and represents the current state of the art for synthesizing assembly language code[88], and show superior performance on more challenging benchmark programs, solving up to 400% more programs within a fixed program evaluation limit and remaining competitive in total performance even when that limit is increased 50x for MCMC.

3.2 Methods

3.2.1 Reinforcement Learning Model of Synthesis

Here I describe the formulation of the general program synthesis task as a multi-step Markov decision process, solvable via reinforcement learning.

In the standard terminology of Sutton & Barto, a fully-observed MDP is a process having some state s_t for timesteps $t \in \{1, 2, \ldots, T\}$. At each state s_t an action from among n possible actions $a_t \in \{1, \ldots, n\}$ is emitted, with some unknown function $p(s_{t+1}|s_t, a_t)$ determining the following state s_{t+1} from among some (typically large) state space. Actions a_t are selected by an agent $A(a_t|s_t; \theta)$ with a policy for selecting actions parametrized by learned parameters θ, commonly a neural network. This policy is trained to maximize the expected cumulative reward value $E(R_T)$ emitted by some reward function $r(s_t, a_t)$, with $R_T = \sum_{t=1}^{T} \gamma^t r(s_t, a_t)$, with time decay factor γ.

Based on these definitions, I represent program synthesis as follows. A *program* of length T is a set of actions $P_T = \{a_1, a_2, \ldots, a_T\}$. Each action $a_{t \in \{1,\ldots,T\}}$ represents a single line of code(e.g. "ADD f0 f1 f2") applied to state s_t, which represents the memory state(the values of all variables) after execution of all previous lines of code $\{a_1, \ldots, a_{t-1}\}$ applied to a set of initial variable assignments. The agent's task, then, is to output the next line of code a_t given s_t such that the final program P_T will maximize a user-provided cumulative reward function R_T on a given set of input and output examples I_j and O_j for a small number of examples j. To fully specify the desired behavior of the program and avoid degenerate solutions that satisfy R but not the programmer's intent, I use multiple input output state pairs, typically 5.

In my instantiation on RISC-V, I allow for multiple input/output examples, so the program state is a tuple consisting of the state of multiple executions. Thus, the initial state is a tuple comprised of all the input states of the examples, and the desired output state is a tuple consisting of all the output states of the examples. I use a reward function combining correctness (distance from current state to the output state) and program length as a metric for computational complexity, but this could easily be extended to include terms optimizing for properties like power consumption, memory usage, network/filesystem IO, or any other desired attribute.

It is worth noting that while this formulation does not by default include non-linear programs containing control flow, it can theoretically be extended to support them, as well as programs containing other elements of modern programming languages as the MDP representation of a process is Turing complete[102], though I leave practical exploration of these topics to further research.

3.2.2 Reward Function

For the reward function $r(s_t, a_t)$, given a set of input-output state pairs (I, O) with N pairs, I use two components. First, a metric for the correctness of the next state for the state s_{t+1} produced from each example I_j compared to the target output state O_j, measured as

$$r_{\text{correctness}}(s_t, a_t) = \frac{\lambda_{\text{correctness}}}{NM} \sum_{j=0}^{N} \sum_{k=0}^{M} \frac{|O[j][k] - s_{t+1}[j][k]|}{|O[j][k]|} \tag{3.1}$$

with M as the number of variables used in this program, and $\lambda_{\text{correctness}}$ as a hyperparameter weight on this component of the reward function. I express correctness as a fraction of $O[j][k]$ to normalize across output values of different magnitudes. To this I add a term penalizing program length to encourage the agent to learn shorter programs as an approximation of program computational efficiency,

$$r_{\text{efficiency}}(s_t, a_t) = |P_t| + 1 \tag{3.2}$$

where $|P_t|$ is the length of the program before taking action a_t. Because the $r_{\text{correctness}}(s_t, a_t)$ term can become large if s_{t+1} is far from O, I combine these terms and scale the resulting values such that large values of $r_{\text{correctness}}(s_t, a_t)$ become close to 0 as

$$r(s_t, a_t) = \frac{\lambda_{\text{scale}}}{r_{\text{correctness}}(s_t, a_t) + r_{\text{efficiency}}(s_t, a_t)} \tag{3.3}$$

with λ_{scale} a hyperparameter weight controlling the scaling of the reward values. I define the space of $(P_t, r(s_t, a_t))$ pairs for a given set of $(I, O, a{\in}\{1, ..., n\})$ as a *program space*, the discrete reward landscape which my RL agent seeks to maximize.

This formulation of program synthesis as an MDP is quite general and can be applied for many different reward functions provided by the user, such as that described above. However, I make a key assumption about the reward function, which is the presence of a (possibly sparse and non-convex) *gradient* in the reward function pointing towards the ground truth program which the RL agent can learn and stochastically descend. While no general guarantee can be made, and indeed it is easy to construct reward functions which provide no gradient,[*] there is evidence for the existence of this gradient for practical program domains and cost functions used in previous work[88,10]. I also demonstrate empirically in my experiments that this gradient is present for many short floating point arithmetic programs in RISC-V.

3.2.3 RL Model

For reinforcement learning, I use a standard dueling double Q-learning algorithm, which learns to predict future rewards for a given state and action using the loss function

$$\delta_t = \|r(s_t, a_t) + \gamma Q_{\theta'}(s_{t+1}, \mathrm{argmax}_{a'} Q_\theta(s_{t+1}, a')) - Q_\theta(s_t, a_t)\|^2 \qquad (3.4)$$

given agent Q-function Q with parameters θ and target Q-function parameters θ', as per Van Hasselt et al.. I set the decay term on future rewards γ to 0.99. The objective of the Q function trained using equation 3.4 is to predict the expected future reward $E(R_t(s_t, a))$ that will result from taking each action

[*]for example, $r(s_t, a_t) = \begin{cases} 1 & \text{if } s_{t+1} = s_o \\ 0 & \text{otherwise} \end{cases}$

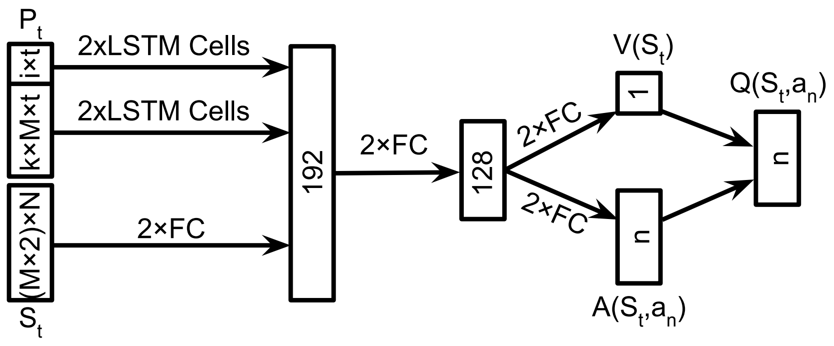


Figure 3.2: Schematic of the Q-function neural network. The input is the memory state S_t of a program P_t of length t plus a set of target memory values for each of M variables for N different examples. The program that produced this memory state is encoded using an embedding for the variables of shape $k \times M \times t$ which is 1-hot in M and for instructions of shape $i \times t$ 1-hot in i given variables passed per instruction k and number of instructions allowed i. These inputs are then processed through the network to compute the Q-function for each state-action pair (S_t, a_n).

$a \in \{1, \ldots, n\}$ possible at state s_t[102]. During synthesis, I then use an ϵ-greedy policy of either taking action $\text{argmax}_a(Q_\theta(s_t, a))$ or else taking a random action with probability $\epsilon = 0.1$.

The input to the agent consists of two parts, the state s_t encoding current and target values for each variable and a sequence of 1-hot vectors encoding previous lines of the program that produced s_t, P_t. A schematic diagram of the network architecture is shown in Figure 3.2. It consists of two input modules, for s_t and P_t. The s_t module is a two-layer fully connected(FC) neural network, while the P_t module is a two-layer LSTM[51] operating on input sequences of up to length p, the maximum program length. These modules are concatenated and fed into two more FC layers, followed by a two-layer value stream computing $V_\theta(s_t)$ the expected future reward value from being in the current state, and a separate two-layer advantage stream computing $A_\theta(s_t, a_n)$ the advantage of

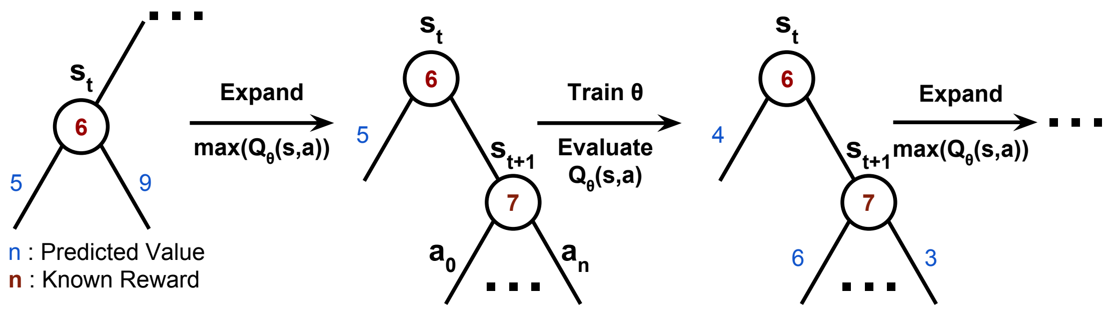


Figure 3.3: Schematic showing how RLGTS combines a Q-function and priority search tree. The Q-function specifies the priority of each unexplored edge in the tree, which then follows an ϵ-greedy sampling strategy. The accumulated experiences from expanding the tree are then used to train the Q-function to improve the prioritization of edges. To reduce the cost of rescoring unexplored edges, the scores are only updated every 100 training iterations.

each action $a \in \{1, \ldots, n\}$ above or below $V_\theta(s_t)$. These modules are combined as per Wang et al. to compute a Q-value per action as

$$Q_\theta(s_t, a_n) = V_\theta(s_t) + A_\theta(s_t, a_n) - \frac{1}{|a_n|} \sum_{j=0}^{|n|} A_\theta(s_t, a_j) \tag{3.5}$$

with each action $a_{j \in \{1,\ldots,n\}}$ representing a single line of code expressible in the language. ReLU non-linearities were used in the FC layers, with LSTM non-linearities following the standard arrangement of sigmoid and tanh functions[51].

3.2.4 Tree Search

Notably unlike most well-studied RL applications, where an agent capable of reliably reaching high reward states across many rollouts of the agent is desired, in program synthesis once the agent discovers a solution to a given program, synthesis is complete and we can stop. I define “solved” here as a candidate program producing $r(s_t, a_t) \geq r_{\mathrm{GT}}$, the reward associated with the output state and GT program length. (In the case where there is no known ground truth solution program one could instead halt the search once a program deemed sufficiently good is found, or after a fixed search time interval). In either case, the ultimate objective is to sample a set of actions leading to a high-scoring state at least once, with repeated samplings of the same action sequence having marginal benefit.

In line with this objective, I combined the Q-function with a simple prioritized search tree algorithm to aid in efficient exploration. The task of the Q-function then becomes to predict a Q-score for each unexplored edge of the tree, the edge representing a possible action $a \in \{1, \ldots, n\}$ to append to the program stored at a node defined by an existing program P_t of some length t. The search algorithm then simply expands $\mathrm{argmax}_{s,a}(Q_\theta(s, a))$, the edge that the current Q-function thinks will yield the highest reward among unexplored edges, evaluating the program $P_t + a$ and adding a new node to the tree with additional unexplored edges. I alternate between sampling unexplored edges(using ϵ-greedy sampling) and training the Q-network, with 100 samples followed by 100 minibatch updates of the network and a rescore of all unexplored edges with the updated Q-network. This process is illustrated in figure 3.3. By taking this approach, I encourage exploration by only evaluating each unique (s, a) combination once, and guarantee worst-case performance to be that of enumerative search, memory and computational resources permitting. While I do not explore it further here, this approach also allows RL-based synthesis to be combined

$$
\begin{array}{rcllllll}
p & ::= & c; p & & & & & \\
 & | & \mathrm{EOF} & & & & & \\
c & ::= & \mathrm{SQRT}\ f\ f & | & \mathrm{ADD}\ f\ f\ f & | & \mathrm{SUB}\ f\ f\ f \\
 & | & \mathrm{MUL}\ f\ f\ f & | & \mathrm{DIV}\ f\ f\ f & | & \mathrm{SGN}\ f\ f\ f \\
 & | & \mathrm{SGNN}\ f\ f\ f & | & \mathrm{SGNX}\ f\ f\ f & | & \mathrm{MIN}\ f\ f\ f \\
 & | & \mathrm{MAX}\ f\ f\ f & | & \mathrm{EQ}\ f\ f\ f & | & \mathrm{LT}\ f\ f\ f \\
 & | & \mathrm{LTE}\ f\ f\ f & | & \mathrm{MADD}\ f\ f\ f\ f & | & \mathrm{MSUB}\ f\ f\ f\ f \\
 & | & \mathrm{NMADD}\ f\ f\ f\ f & | & \mathrm{NMSUB}\ f\ f\ f\ f & & \\
f & ::= & \mathrm{f1}\ |\ \ldots\ |\ \mathrm{f32} & & & & &
\end{array}
$$

Figure 3.4: Grammar describing the subset of RISC-V that RLGTS synthesizes.

easily with search-based synthesis methods using deductive reasoning and search tree pruning for DSL-specific performance improvements.

3.3 Experiments

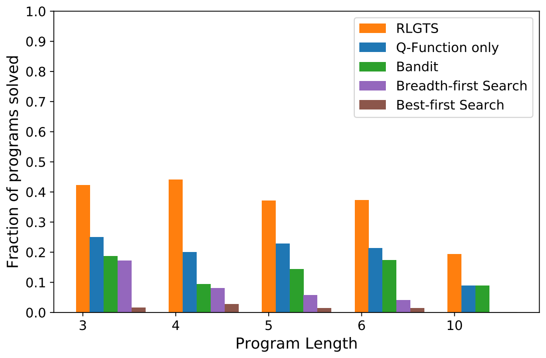
3.3.1 Experiment Setup

To analyze the performance of my instantiation of RLGTS, I synthesize programs using a common subset of the RISC-V assembly programming language[114]. I select a core subset of instructions on floating point values, shown in Figure 3.4, which excludes control flow, memory reads/writes, and "magic number" inputs for simplicity. This domain is interesting because there are few previously published methods that can perform better than naive enumerative search other than stochastic search using hand-tuned MCMC search to estimate the program gradient[88].

To benchmark performance, I construct a dataset of synthetic programs by generating random programs with specified attributes. For each program in the evaluation set, I specify the number of lines, number of allowed instructions, number of allowed variables, and number of examples, and synthesize a random program to satisfy these specifications, rejecting programs which can easily be expressed in fewer lines. This generation process allows the use of fewer types of instructions or fewer variables, but will always obey the length and example count specification exactly, and each method must search the entire program space defined by the specified constraints. I generated 100 programs for each set of specified attributes. The resulting search spaces have sizes ranging from 2.1×10^6 possible programs (3 lines, 2 instructions, 4 variables), to 4.0×10^{31} possible programs (15 lines, 2 instructions, 4 variables).

To train the network, I use a learning rate of 0.001, and sampled batches of 64 experiences from an experience buffer storing all previously observed states using proportional prioritized experience replay as per Schaul et al., with $\alpha = 0.6$ and $\beta_{\text{initial}} = 0.5$, linearly annealing β to 1.0 after 10,000 iterations. I updated the target Q network parameters by setting $Q_{\theta'} = Q_\theta$ every 100 training iterations. I set the reward function hyperparameters $\lambda_{\text{correctness}}$ and λ_{scale} to 5 and 100 respectively for all methods. While this system contains a number of hyperparameters that affect performance, I performed only basic manual parameter tuning on a short hand-written test program intended to be readily solvable, and a set of 10 randomly generated length 3 programs which were not used to generate results, with the resulting parameters used for all experiments.

Each method was allowed to run on each program in the benchmark suite until a satisfying program was found, 20,000 programs were proposed, 16 GB



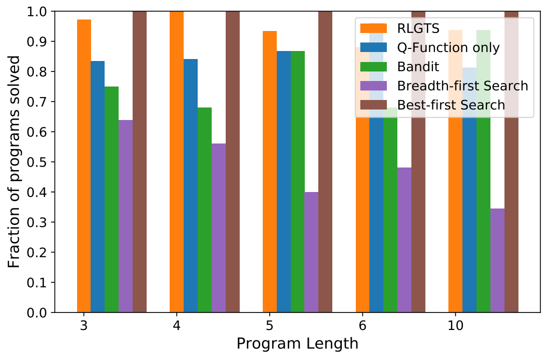


Figure 3.5: Comparison of performance between RLGTS and baseline methods, showing fraction of non-convex programs solved in the left figure and fraction of convex programs solved in the right figure. While RLGTS is effective at solving convex programs, they can be easily solved via best-first search, and thus I remove them from the other experiments as they are not a good indicator of method performance.

of memory were consumed, or 8 hours on one 2.4GHz Broadwell CPU and one Nvidia p100 GPU had been consumed. In my comparisons, I focus on sample efficiency instead of clock time, and while proposing 20,000 programs takes considerably less time and computation using purely search based methods running on CPU, both neural network and search-based methods could be further optimized to increase speed. Because of this, I do not consider wall clock time to be definitive for any method here described and thus focus on the number of proposed programs required to solve instead as a measure of efficiency and the potential of the method. I expect with further optimizations and additional hardware that much harder programs could be tractably solved by RLGTS.

3.3.2 Baselines

I compared RLGTS to several baselines. In the simplest case, I ran a breadth-first enumerative search algorithm, which has expected program solve time of approximately $N/2$ for a program space with N possible programs of length equal to the GT program.

Second, I compared to a simple multi-armed bandit model[58] as the simplest form of RL-trainable model, representing each decision x defining a program as an independent random variable sampled based on a learned probability $p_\theta(x)$, trained using the REINFORCE algorithm[115].

Next, I compared to a Q-learning baseline as an ablation of the full system, using the same network architecture and training procedure, but lacking the search tree of the full system.

Lastly, I compared to the approach used in Stoke, a heuristic-driven stochastic search based system and currently the state of the art for optimal synthesis of RISC-V programs[88]. To accommodate the simplifications I made to the RISC-V language, I re-implemented Stoke's MCMC search to allow it to search over the RISC-V space and reward function. I refer to this baseline as "MCMC." I selected a value for the MCMC β hyperparameter by testing on a holdout validation program, and found that $\beta = 2$ works well.

3.3.3 Program Length

Because program length is a major determinant of search space complexity, I characterized each method's behavior as a function of the length of ground truth program. Figure 3.5 shows the results on my synthetic benchmark for lengths between 3 and 10 lines. RLGTS solves at least twice as many programs as the best baseline, the bandit. The addition of the priority search tree improves performance over the Q-function-only ablation by 70-100% consistently.

A fraction of the programs generated have a convex reward function space and can be trivially solved by convex descent using a best-first search algorithm. Specifically, for a program to be considered convex, among the rewards of all possible length 1 programs the first line of a satisfying program ranks highest, among all length 2 extensions of that best length 1 program a length 2 subprogram of a satisfying program is the highest ranked, and so on until a satisfying program is found. I found in testing that about 30% of programs I generate have this property. As these cases are easily solvable via a naive best-first search algorithm and are expected to be rare among programs of human interest, I filter and remove them from the remainder of the evaluation. Success rates on convex programs are included separately in figure 3.5, where RLGTS solves more than 90% of such cases, albeit at the cost of more computation than best-first search.

Search Depth Limit

While RLGTS readily outstrips the naive search and multi-armed bandit baselines for various lengths and search depths, I found during testing that MCMC search performance on this benchmark is highly dependent on the difference between the true program length and the search depth limit on maximum program length I place. This is consistent with its original proposed use-case

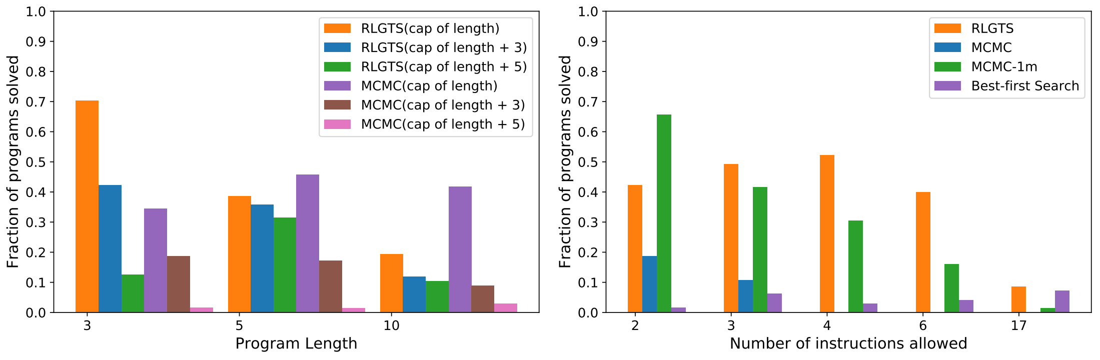


Figure 3.6: Comparison of performance between RLGTS and MCMC as a function of program length (left) and number of instructions allowed (right). RLGTS retains better performance as the difference between program length and the search cap increases. Surprisingly, MCMC performance decreases only slightly if at all as program length increases. MCMC sample efficiency drops off much more rapidly as the number of instructions searched over increases.

of super-optimization, wherein an existing program is provided as part of the specification, which allows for a good bound on search depth to be defined[88]. Because of this sensitivity, I compare RLGTS against MCMC for fixed differences between ground-truth program length and maximum search depth, The results of this comparison are shown in figure 3.6. While MCMC performs competitively with RLGTS on longer programs when the target program length is known, its performance degrades rapidly when the maximum program length diverges from it, losing 50-80% of its performance when the cap is 3 lines above the GT length, and is effectively 0 at length + 5. While RLGTS is also adversely affected by not knowing the GT program length, the impact is smaller, in the range of 10-50%.

3.3.4 Action Space Complexity

Lastly, I investigated how performance drops off as the number of instructions increases. Figure 3.6 shows performance for RLGTS and MCMC, as well as best-first search, as a function of the number of instructions, ranging from 2 instructions to all 17 instructions. All programs are of length 3, with a maximum search depth of 6. RLGTS solves between 40-50% of all programs with few instructions, and performance does not drop off until all 17 instructions are used. In contrast, the performance of MCMC falls off much more rapidly and its success rate is 0% on 4 or more instructions with a limit of 20,000 attempts. In response, I ran MCMC with a more generous limit of 1,000,000 attempts. With a limit 50 times higher, MCMC is able to exceed RLGTS performance on programs of 2 instructions and match it on 3. MCMC is still unable to solve any programs allowing all 17 instructions, however, while RLGTS solves around 9%.

3.4 Discussion

Here, I have presented reinforcement learning guided tree search, a new approach for program synthesis powered by reinforcement learning. This approach is general and flexible, with up to 400% better performance than the state of the art in traditional search-based methods on cases where search achieves a non-trivial success rate. Its performance suggests that with further research RLGTS may be able to scale to solve more complex programs that cannot be solved by previous methods.

4 Conclusion

The sampling challenges described in this thesis only begin to address the topic. While in all likelihood we will never attain "optimal" sampling in which only the minimal data needed is sampled, further progress towards that ideal is possible. In the following sections, I'll discuss future directions building on each chapter, followed by some discussion of other interesting directions on sampling from the recent literature.

4.1 Continuous-Action Optimization

The topic of sampling optimal actions has some surprising complexity. Even on the scale of single actions we still see an exploration versus exploitation tradeoff, where methods must inject some amount of noise (as in DDPG and TD3) or have an innately noisy policy (as in Qt-Opt and CGP) in order to avoid getting stuck in a deterministic local maxima in the action landscape. This issue also appears for on-policy methods– while common policy gradient algorithms such as PPO[93] use stochastic policies which parametrize a distribution over actions, good hyperparameters are needed to avoid the policy collapsing to a near-deterministic distribution. While CGP neatly solves the issues of inference-time efficiency with hyperparameter robustness, several extensions would improve on the method:

- **Better sample-based optimizers** would reduce the performance penalty of CGP relative to TD3/SAC. One promising approach is to use Covariance Matrix Adaptation (CMA)[48] in place of CEM. CMA is commonly used in the evolutionary algorithms community[107,35], and while it outperforms CEM it is more expensive to compute. As CGP assumes that compute costs at training time are not a limiting factor, CMA is suitable where it might not be without the neural network policy for use at inference time. I performed some preliminary experiments on this direction, and found that while training becomes much slower the inference-time performance of CGP using CMA matches or exceeds that of TD3/SAC.

- **Regressing the output of sample-based optimization tasks** in other domains could also prove useful. One example of a possible extension is program synthesis– sample-based optimization methods such as Stoke[88] are among the most promising and general non-ML-based program synthesis methods, but are expensive to compute. Training a neural network to predict the output of Stoke given its input could serve as a useful shortcut for many programming tasks. Other applications also present themselves– a CGP-style inference-time approximator network could be trained for any task where Monte Carlo sampling or sample-based evolutionary optimization is effective but computationally prohibitive for some use cases. Examples can be found in many modeling applications throughout the physical sciences.

- **Learning a sample-based optimizer** is another interesting direction– rather than use a generic optimization algorithm such as CEM or CMA, it is possible to train a neural network using an RL objective function to act as a domain-specific sample-based optimizer, which may be more sample efficient and more accurate than traditional sample-based optimizers. Some examples of this approach for other domains include work by Mirhoseini et al. as well as chapter 3 of this thesis, but the same approach could be applied for optimizing other black box functions such as Q-function actions. As deep neural network optimization landscapes have a number of unique features depending on architectural details (for example, batch norm[85]), it is reasonable that a sample-based optimizer trained to optimize neural network inputs would discover sampling strategies that are more efficient on neural networks of a given architecture than a general-purpose baseline, similar to other domains like program synthesis and chip floorplanning[72]. Related to learned sample-based optimizers, work on learned gradient-based optimizers for neural networks has also been described, though a number of challenges remain for outperforming hand-crafted algorithms[70].

4.2 Reward Prediction Error Exploration

Building upon the concept of separate exploration and exploitation policies developed in Chapter 1, I next described a split-policy approach for long-time-horizon exploration in Chapter 2. This method is interesting in several ways: It splits the steps of exploration and exploitation into separate agents, each learning independently and communicating only through a shared replay buffer. It yields an exploration objective dissimilar to existing exploration objectives, which is both reward-aware and analogous to some biological exploration phenomena. That exploration objective also shows promise for solving a number of diverse tasks of different types, where other objectives are not suitable. That said, the QXplore as a first attempt at using reward prediction error to explore does suffer from some limitations: Noisy rewards can serve as a distractor, and QXplore will learn slowly on tasks where there are many ways to fail (yielding worse-than-expected reward) but few ways to succeed (yielding better-than-expected reward). Some future directions to address these issues, as well as other interesting questions, include:

- **Signed Reward Prediction Error** could be used to focus exploration on better-than-expected returns, which in theory addresses both issues raised above. However, in preliminary experiments this approach underperforms the unsigned RPE, likely because most of the magnitude of bootstrapped Q-function TD-error is determined by the disconnect in predictions between the target Q-function and current Q-function rather than between the Q-value and the empirical return. This approach might work better for Q-learning without a target Q-function, or for learning with a non-bootstrapped value function, such as used by on-policy actor-critic methods.

- **Better measures of uncertainty and surprise about rewards** can certainly be formulated. In the exploration literature, much effort has gone into developing better measures of the novelty of a given state or trajectory which can better motivate exploration. Naive value function RPE is one method of estimating return novelty, but others exist. As one example, the concept of Credit as a measure of the mutual information between an action and a future outcome could be used for exploration by focusing sampling on trajectories yielding high MI between actions and future rewards. As high-credit trajectories are inherently useful for learning (I have some work not in this thesis using credit as a post-sampling threshold to subsample on-policy data showing this effect[4]), an exploration policy focusing on high-credit data should be effective at sampling useful data and enabling learning on tasks with complex reward landscapes.

- **Investigation into the analogy between RPE and biological exploration** could prove fruitful as well. In Chapter 2, I observe that the Q_x agent learns behaviors analogous to boredom in animals, wherein it probes the boundry between rewarding and non-rewarding states in elaborate ways (see Section 2.5.4 for more details). As prior work has shown that reward prediction error is analogous to changes in activity by dopaminergic neurons in the brain[75,104], RPE-seeking is then analogous to dopamine-seeking, which may drive human and animal behavior and contribute to some types of addictive behavior[82]. As the mammalian dopamine system is considerably more complex than a single monolithic value function, and is known to track environment variables other than a biological analog to reward[24,15,68], following this analogy into the neuroscience literature may yield additional insights into how we can provide intrinsic motivation for RL agents to explore efficiently.

4.3 Q-Function Prioritized Tree Search

In Chapter 3, I diverged from the previous two chapters to discuss sampling in the case where we have a known transition function and a discrete state space, but the number of states is uncountably vast and the reward landscape is not well-shaped. Traditionally this has been the domain of search algorithms, but in Chapter 3 I showed how a hybrid approach using both RL and search can outperform either method individually in sample complexity.

This approach shows promise, but in this thesis I have only scratched the surface of what is possible, both for program synthesis in specific and structured exploration in general. Other recent work has also explored these topics, perhaps the most notable of which is Go-Explore[23], which uses a similar RL+search approach for Atari games in combination with a compression algorithm to alias states into a tractable number of state clusters. Below, I will discuss some other possible extensions to my work:

- **Improving inter-program generalization** should be a major topic of future research in this area. RLGTS as described here is only capable of learning to search for one program at a time, and my preliminary attempts to generalize across programs performed worse than training from scratch. One limitation here is the state representation– each program is specified as a set of (random) input and output numbers, and generalizing to new programs requires learning to generalize across arbitrary floating point values. A consistent input/output specification such as a formatted text description or examples using canonical inputs would make this problem easier to learn.

- Another challenge is to **improve the diversity of training data**– even with a limited-size vocabulary of instructions and registers the range of possible programs is vast, and sufficient coverage of this space via generating random synthetic programs is intractable. One interesting observation from this work is that the set of human-relevant programs appears to occupy only a small subspace of all possible programs– most of the synthetic programs I test on do not evaluate useful functions. Biasing the set of programs considered based on human relevance, such as by pretraining via imitation learning on human-written code or a constrained training set should both help the generalization of the agent as well as allow for synthesis of more complex programs. Notably, synthesis via human program imitation has recently been put into production by Github and OpenAI as their Copilot product, in closed beta testing as of the time of writing. This system is consists of a large language model trained on human-written code acting as a "stochastic parrot" without any consideration of the correctness or validity of the output (which is delagated to the human user). Combining such a language model pretraining step with feedback on validity and correctness from an interpreter would likely improve upon the imitation-only approach now in use.

- **RL+Search in other domains** could also prove useful. Some work has previously been done here, such as Go-Explore[23] for Atari games, but other problem domains remain unexplored. One possible domain that has gotten some attention recently is RL for chip floorplanning[72], the process of laying out elements in an integrated circuit design to optimize various metrics (such as power and space) without violating physical constraints. Previous work has shown that RL can provide useful solutions to this problem, but as the state space is vast an approach like RLGTS could improve performance. Similarly, search in combination with an RL objective could be used for interactive text-based tasks, such as text games. Other emerging applications of RL such as CAD design[94] and molecular design[95] also have the necessary properties to allow for an RL+search approach, and could benefit as well. Broadly, environments with known transition functions where an optimal trajectory cannot be inferred from the initial observation should benefit from the backtracking and prioritization that RL+search provides.

4.4 Future Directions

This brings us to the end of this thesis. The work I have described here spans a range of aspects of sampling, but is not the final word on the topic. Many questions remain: How can we define what "efficient sampling" means? Is there a unifying approach to sampling that applies across domains other than return maximization? How do the three pillars of an RL algorithm– learning rule, agent architecture, and sampling method– interact in common domains? Can strength in one or two pillars cover for weakness in a third? How can we explore (sample) more efficiently in tasks that don't require long-range exploration? In this section, I will discuss some interesting directions and related work that I hope will drive the field going forward.

4.4.1 The Three Pillars of RL

Any learning system has three major components– The update rule or objective function, the learning agent with parameters to be trained, and the data to learn on. The nature of these components will vary depending on the field, as will the importance placed on them. For example, in modern natural language processing neural network architectures (the learning agent) have been the subject of much research, and the most powerful learning systems use transformer networks with billions of parameters[12]. These large models are trained on vast datasets with billions of elements, but the update rules are comparatively simple, and are not a major topic of research.

In contrast, in reinforcement learning update rules have been the primary focus, and most RL papers describe new objective functions for learning RL tasks. While data (which is sampled/generated during training, in the context of RL) has received some research attention, including in this thesis, the learning agents used in RL are typically simple and don't vary much between papers on a given task. For example, the relatively small neural network architecture described by Mnih et al. in 2015 remains in use by most authors working on the Atari benchmark today[4]. Only recently has much attention been paid to the question of neural network architectures for RL, and there has been some promising recent work on the topic[77]. Looking forward, combining enhancements in all three– update rule, sampling method, and agent architecture– will likely lead to real improvements in the performance of RL algorithms.

4.4.2 Self-Supervised Learning

In addition to improving online sampling, it is also possible to use other sources of data to train RL algorithms. *Self-Supervised learning* in the context of RL is typically an interactive training phase in which the agent collects interactions using some reward function other than the eventual task reward. This includes classical exploration performed as a pretraining step[67,118,86], but also includes things like training to reach self-selected goal states[6,80], automatic generation of parametric tasks[111], and self-play[29,26,25]. The idea in each case is to provide a reward function that generalizes the eventual task reward and avoids the need for extensive shaping of reward functions or direct human supervision.

The major question for such methods is how well the proposed self-supervised objective transfers to the eventual human-provided objective, and for which types of tasks this is true. While self-supervised RL has shown some promising progress, these methods have largely only been demonstrated in a few domains (robotics) and with relatively simple environments where new human-provided goals are similar to self-supervised goals. For more complex tasks in robotics and elsewhere it will prove intractable to cover the entire state space in this way (e.g. a robot that must function in any user's home anywhere on earth), and so stronger analogy-making will be needed (reaching into one cabinet must resemble any other cabinet). Further, these methods rely on long self-supervised pre-training stages where the RL agent can sample many trajectories (far more than needed to learn a single variant of the task), making them unsuitable currently for domains where sampling is expensive.

4.4.3 Offline RL

In addition to self-supervised learning, *Offline RL* has also seen some recent successes, with a number of high-profile publications in the field[17,62,64]. In offline RL (and the closely related area of *imitation learning*), initial pretraining is performed on a non-interactive dataset of trajectories sampled by some other agent, such as a baseline algorithm or a human. The goal in this case is to learn the task being performed by the agent that collected the data prior to further online training or inference using the learned policy. The advantage here is clear– by using a prior policy that is known to perform the task "good enough" (and which is known to be safe, in the case of safety-critical tasks like autonomous driving or healthcare decision making), the initial learning and exploration phase using a near-random policy can be avoided, resulting in many fewer trajectories needed to learn the task.

The major challenges for offline RL currently center on the process of transferring from pre-collected data to live agent. While recent work such as conservative Q learning[62] has found success in obtaining stable policy behavior when faced with states not seen by the data policy, issues of domain mismatch still remain– if the offline data is sufficiently different (for example, if it was collected from a human POV) from the data observed by the agent at inference time then good behavior is not guaranteed. Similarly, for some domains it may be difficult or impossible to collect much offline data, so transfer from related tasks will be necessary. Developing good methods for transferring and fine-tuning these offline trained models is likely to be a major direction of research going forward.

4.5 Final Thoughts

Sampling in reinforcement learning is a very broad topic, and touches on many aspects of the field. While much more could be said on the topic, and much work remains to be done, I will conclude this thesis with some high-level take-aways on sampling and exploration:

- Intelligent sampling methods can affect the tractability and learning stability of many MDPs. Changes to sampling methods can have an outsize impact compared to changes to RL update rules or agent architectures.
- The benefits of a given sampling method are often domain-specific and depend on the properties of the MDP to be solved. Successful use of RL to solve real tasks requires reasoning about these properties, and sampling methods are an important part of the RL toolbox for overcoming them. Taking advantage of inductive biases in specific contexts can greatly improve performance.
- Similarly, borrowing techniques from outside the deep RL literature can benefit sampling. In addition to sample-based optimizers and search algorithms, this can also be seen in offline RL, which borrows techniques from supervised learning to overcome the limitations of a fixed dataset. Con-

cerns about data distributions and sampling are not unique to RL, and many ideas from other fields could be useful for RL as well.

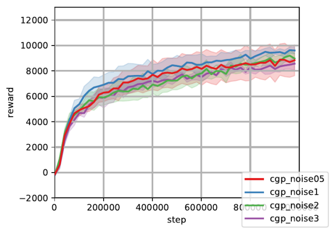
With that, I conclude this thesis. It is my sincere hope that you, the reader, have gained some useful insights into sampling and exploration in reinforcement learning, and that you will carry them forward to inform your own efforts. Thank you for reading. If you have further questions, I can be reached at rileys@cs.princeton.edu.

A

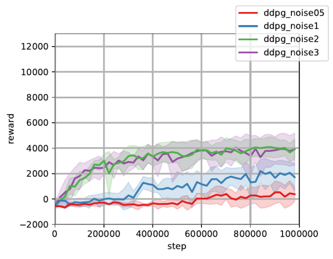
Supplemental Materials to Chapter 1

A.1 Detailed Method Stability Analysis

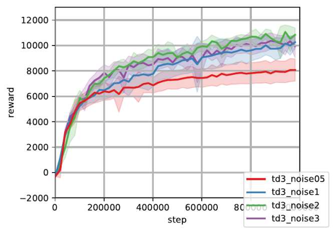
As part of my exploration of method stability, I ran a battery of hyperparameter sweeps on the HalfCheetah-v2 benchmark task. See the results in Figures 1, 2, 3, 4 and 5.



(a) CGP

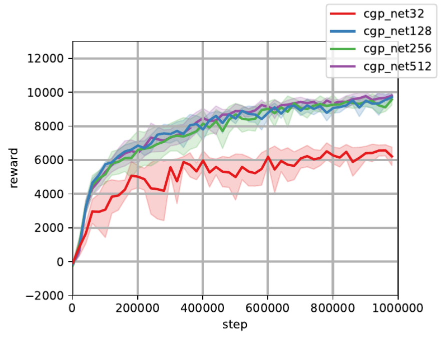


(b) DDPG

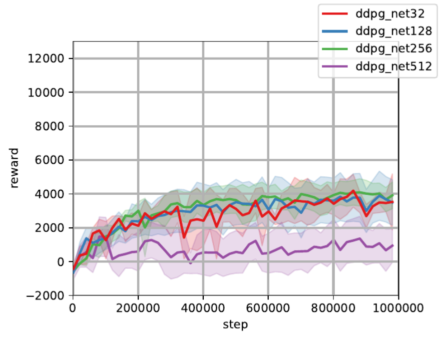


(c) TD3

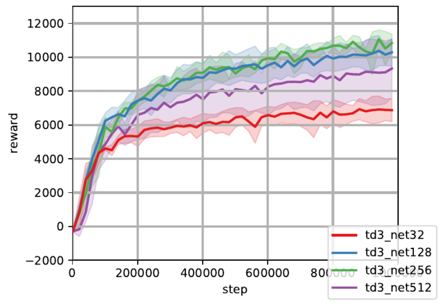
Figure A.1: Sensitivity of various methods (CGP, DDPG, and TD3) to variations in noise parameters. These methods all use noise as part of their specification. I varied all sources of noise on the discrete interval {0.05, 0.1, 0.2, 0.3}. Both CGP and TD3 can tolerate variations in noise well, but TD3 performance falls off when noise is reduced, as they require sufficient noise for sampling diverse training data.



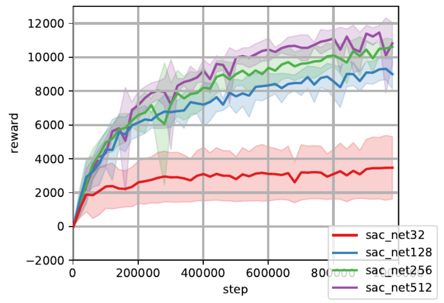
(a) CGP



(b) DDPG

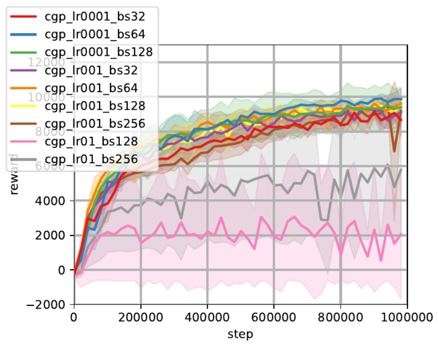


(c) TD3

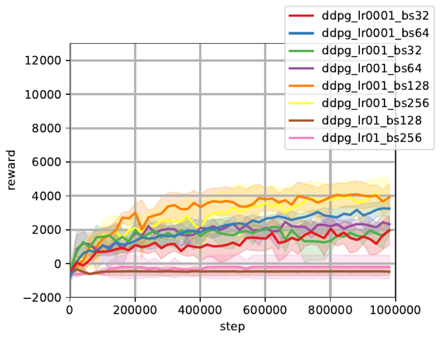


(d) SAC

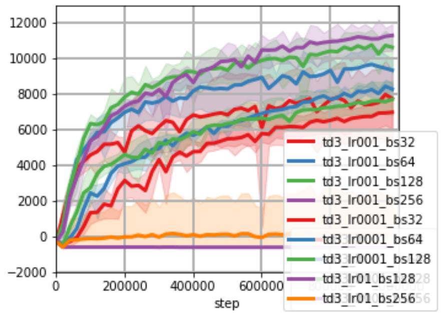
Figure A.2: Sensitivity of various methods (CGP, DDPG, TD3, and SAC) to differences in number of units in their fully-connected layers for both Q-function and policy network. I varied all layer sizes on the discrete interval {32, 128, 256, 512}. All methods other than DDPG degrade in performance with a 32 size network, but CGP is affected much less by large/small networks outside that extreme.



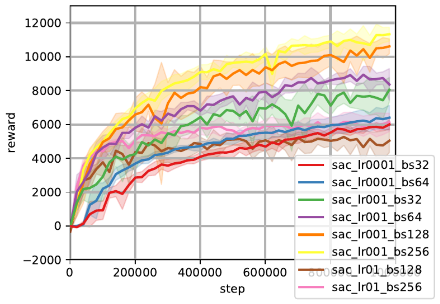
(a) CGP



(b) DDPG

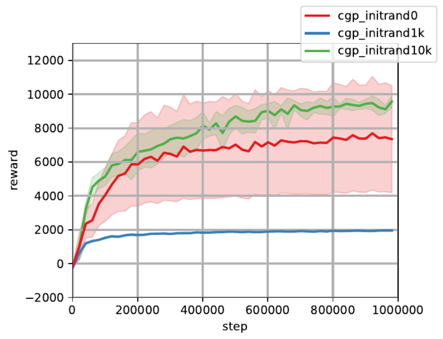


(c) TD3

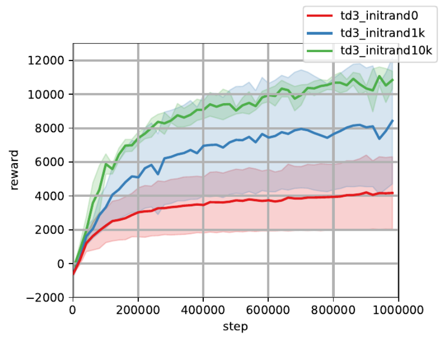


(d) SAC

Figure A.3: Sensitivity of various methods (CGP, DDPG, TD3, and SAC) to different combinations of learning rate and batch size. I varied all layer sizes on the discrete learning rate interval {0.0001, 0.001, 0.01} and the discrete batch size interval {32, 64, 128, 256}. The performance spread across learning rates and batch sizes is much narrower for CGP compared to other methods, with the exception of a learning rate of 0.01, which was too high for CGP to stably train.

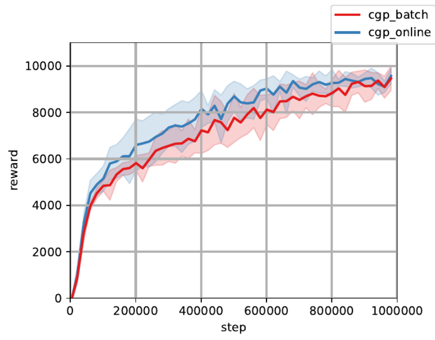


(a) CGP

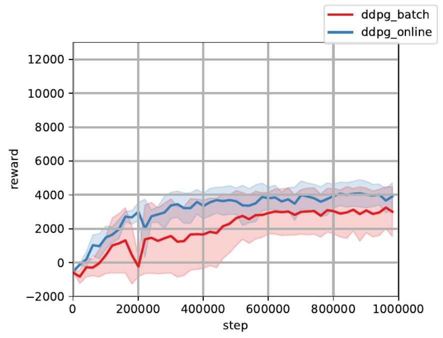


(b) TD3

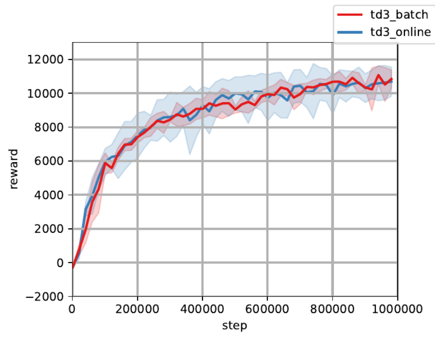
Figure A.4: Sensitivity of two methods (CGP, and TD3) to differences in number of random samples of the action space seeding the replay buffer. I varied the number of random steps on the discrete interval $\{0, 1000, 10000\}$. As described by the TD3 authors, seeding the buffer is crucial to performance. For runs with lower random sample counts, the agent gets stuck in a local reward maxima of around 2000 with some decent probability.



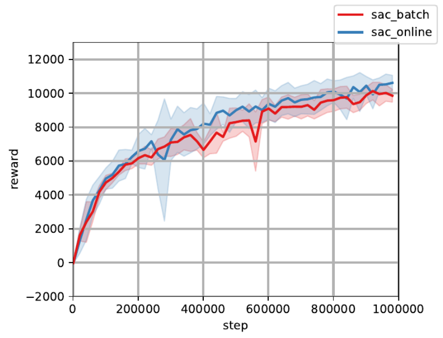
(a) CGP



(b) DDPG



(c) TD3



(d) SAC

Figure A.5: Sensitivity of various methods (CGP, DDPG, TD3, and SAC) to training procedure of training online vs. offline. Online training is defined as, after each step through the environment, training the Q function for at least 1 step and potentially updating the policy. Offline training is defined as rolling out the policy uninterrupted for a full episode, and then training the Q function for a fixed number of steps and/or updating the policy. Only DDPG experiences significantly worse performance in one mode or the other, though online training usually performs slightly better for CGP and SAC.

Table A.1: Hyperparameters used for CGP benchmarking runs.

Hyperparameters	
CEM	
iterations	2
sample size	64
top k	6
Networks	
num units	256
Training	
policy lr	0.001
q lr	0.001
batch size	128
discount (γ)	0.99
weight decay	0
soft target update (τ)	0.005
target update freq	2
Noise	
policy noise	0.2
noise clip	0.5
exploration noise	0.0

A.2 Hyperparameters

To facilitate reproduce-ability, the hyperparameters used are listed in Table A.1.

A.3 Implementation Details

One critical implementation detail that was found to dramatically affect performance is the handling of the *done* state at the end of the system during a training episode. The OpenAI Gym environment will emit a Boolean value *done* which indicates whether an episode is completed. This variable can signal one of two things: the episode cannot continue for physical reasons (i.e. a pendulum has fallen past an unrecoverable angle), or the episode has exceeded its maximum length specified in a configuration. Since the first case is Markovian (in that it depends exclusively on the state when *done* is emitted, with current timestep not considered part of the state for these tasks), it can safely indicate that the value of the next state evaluated at training time can be ignored. However, in the second case the process is non-Markovian (meaning it is independent of the state, assuming time remaining is not injected into the observation state), and if the *done* value is used to ignore the next state in the case where the episode has ended for timing reasons, the Q-function learned will reflect this seemingly stochastic drop in reward for arbitrary states, and policies sampling this Q function (either learned or induced) will empirically perform 20-30% worse in terms of final reward achieved.

To resolve this, for tasks which are non-Markovian in nature (in this paper, HalfCheetah-v2 and Pusher-v2), I do not use a *done* signal, which means that from the perspective of the Q-function the task has an infinite time horizon, where future states outside the time limit are considered as part of the Q-function but never experienced.

For TD3[*] and soft actor critic[†], I used the author's published implementations in my experiments, combined with my outer loop training code to ensure the training process is consistent across all methods.

Experiments are run with Python 3.6.7. Critical Python packages include

[*]https://github.com/sfujim/TD3
[†]https://github.com/vitchyr/rlkit

`torch==1.0.0`, `numpy==1.16.0`, `mujoco-py==1.50.1.68`, and `gym==0.10.9`. I used MuJoCo Pro version 1.50 as the simulator. Performance benchmarks were measured on workstations with a single 24-core Intel 7920X processors and four GTX 1080Ti GPUs.

B

Supplemental Materials to Chapter 2

B.1 Implementation Details and Hyperparameters

I describe here the details of my implementation and training parameters. I held these factors constant and used a shared codebase for QXplore, RND, and ϵ-greedy to enable a fair comparison. I used an off-policy Q-learning method based off of TD3[33] and CGP[98] with twin Q-functions and a cross-entropy method policy for better hyperparameter robustness. Each network (Q_θ, $Q_{x,\phi}$, RND's random and predictor networks) consisted of a 4-layer MLP of 256 neurons per hidden layer, with ReLU non-linearities. I used a batch size of 128 and learning rate of 0.001, and for QXplore sampled training batches for Q and Q_x of 75% self-collected data and 25% data collected by the other Q-function's policy as described in Algorithm 3.

For DORA[30], I used the hyperparameters and training procedure specified by the original paper where possible, though it was necessary to adapt the method somewhat to the continuous action domain. This is because the original formulation proscribed an "LLL" action selection scheme that requires taking discrete log-probabilities of the distribution of Q and E values over actions, which is not tractable in continuous action spaces. Instead, I tried selecting actions using either a CEM policy that maximizes the sum of the two objectives, or using the E values as a reward bonus for training Q and selecting actions that maximize Q only. I thus expect the performance of my implementations to be somewhat worse than a hypothetical distributional-DORA, though the action selection scheme I used does make this version directly comparable to QXplore and RND. Both formulations behaved similarly on `SparseHalfCheetah` and did not achieve reward with any frequency.

For ϵ-greedy sampling with continuous actions, I sampled a uniform distribution of the valid action range (-1 to 1 for all tasks) with probability ϵ and act greedily otherwise. I note that the stochastic cross-entropy method policies I used for all experiments also introduce some amount of local exploration through noisy action selection.

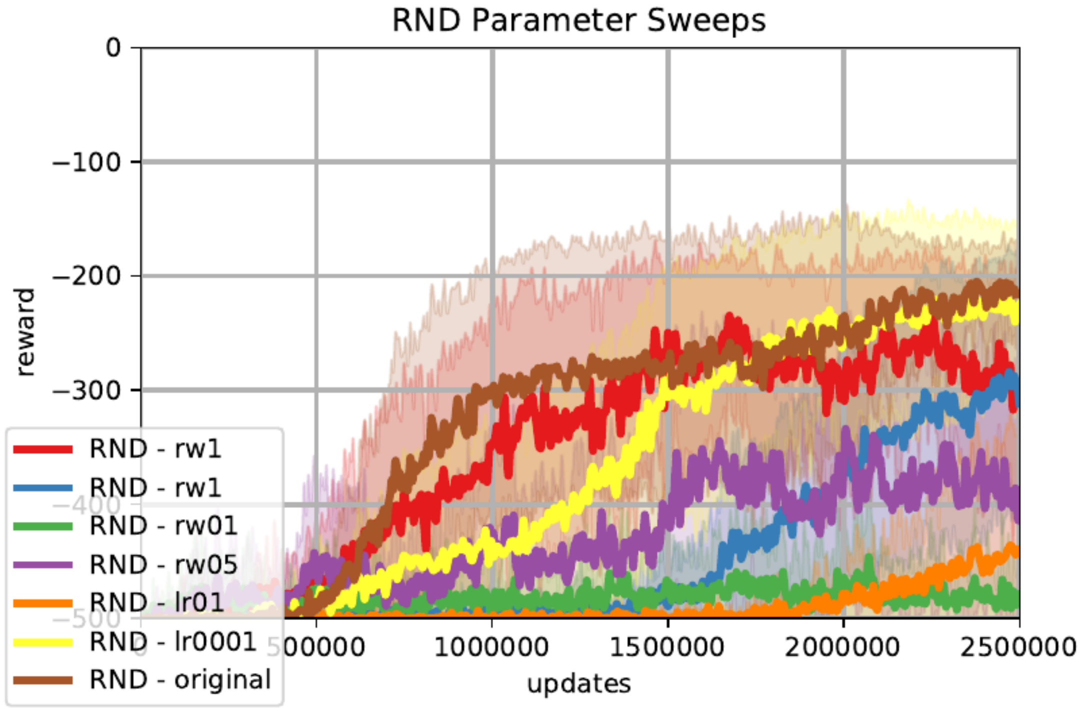
The parameters I used for the benchmark tasks are shown in Table B.1.

B.1.1 RND Parameter Sweeps

As I have adapted RND to operate with vector observations and continuous actions, I performed several hyperparameter sweeps to ensure a fair comparison. I report in Figure B.1 the results of varying both predictor network learning rate "lr" and extrinsic reward weight "rw" independently on the `SparseHalfCheetah` task. The baseline values for these parameters used elsewhere are 0.001 and 2 respectively. I observe that RND is fairly sensitive to reward weight, but a value of 1 or two performs well, while a learning rate of 0.001 appears to learn faster early in training without loss of final performance.

B.2 The 'Noisy TV' Problem

The 'Noisy TV' problem is a classic issue with some state-novelty exploration methods in which states with unpredictable observations serve as maxima in the novelty reward space. QXplore's TD-error objective is not fundamentally vulnerable to the problem, but to demonstrate that my function approximation early in training is also not subject to it, I trained QXplore on a variant of the `SparseHalfCheetah` task where I add a random normally-distributed value to the observation vector of the agent. The variance of this noise value increases proportionately to the movement of the cheetah in the negative direction (away from the reward threshold). An agent vulnerable to the noisy tv problem will



/2/2

Figure B.1: Parameter sweeps for RND. A reward weight of either 1 or 2 works best, with a learning rate of 0.0001 a close second.

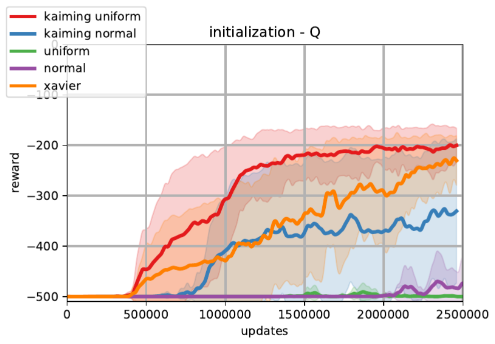
be enticed to explore in the negative direction rather than forward, as this maximizes the novelty/unpredictability of the observations.

I show the results of training QXplore on this environment in Figure B.3 for both Q and Q_x, as well as the mean position of the cheetah along the movement dimension during Q_x's training rollouts. As expected, the performance of neither Q nor Q_x is meaningfully altered relative to the baseline, and Q_x is not biased to explore backwards to a greater degree than it typically does early in training.

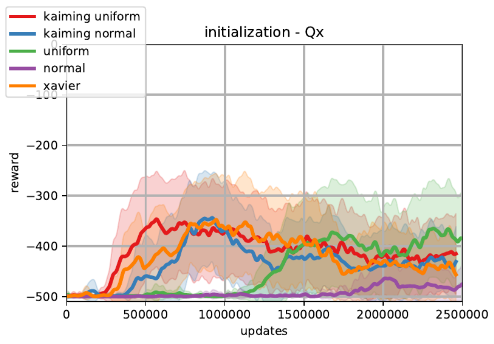
B.3 Weight Initialization

As I use neural net function approximation error as a state novelty baseline for early exploration, the behavior of Q_x may be sensitive to weight initialization. To test this, in addition to the Pytorch default initialization method "Kaiming-Uniform,"[49] which I used for all runs outside this section, I also tested initializing both Q and Q_x with "Kaiming-Normal" and "Xavier-Uniform,"[38] two other initialization methods that result in higher variance between initial outputs of the networks, which translates into reduced training performance. I further tested two naive distributions that produced very high variance in outputs, "Normal," sampling weight values from $\mathbf{N}(0,1)$ and "Uniform," sampling values from $\mathbf{U}(-1,1)$. These configurations were not expected to perform as well as "Kaiming-Uniform", but do test the ability of Q_x to explore given a poor initialization. In all cases other than "Kaiming-Uniform" I set the bias of each neuron to 0. The results of QXplore with each initialization scheme on `SparseHalfCheetah` are shown in Figure B.2.

"Kaiming-Normal" and "Xavier-Uniform" both showed moderate decrease in overall performance, though both Q and Q_x were able to converge on reward. "Normal" and "Uniform" however both more-or-less prevented Q from converging on reward. Their effect on the ability of Q_x to find reward however is much more mild- only "Normal" and to a lesser extent "Uniform" caused significant issues with discovering and converging on reward. This suggests that Q_x is not particularly dependent on careful weight initialization to explore with function approximation error.

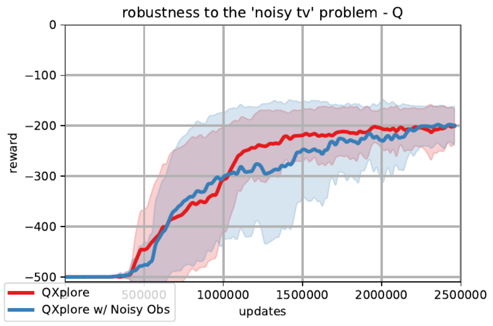


(a) Initialization of Q

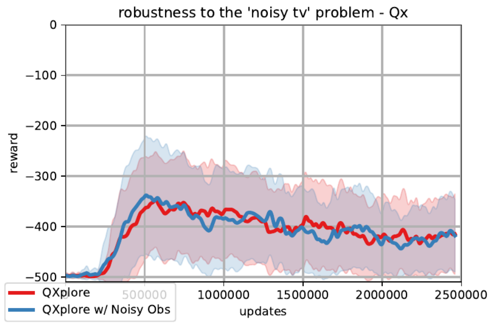


(b) Initialization of Q_x

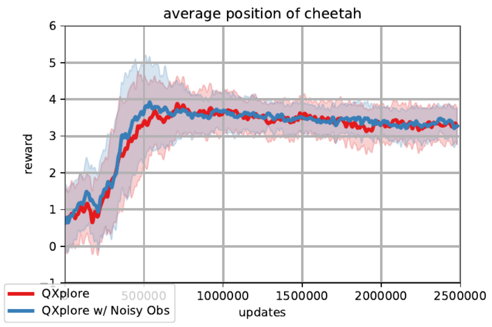
Figure B.2: Several alternate initialization schemes for Q and Q_x. While Q is adversely impacted, Q_x is relatively robust even to very poor initializations such as "Normal" and "Uniform."



(a) Noisy observation effects on Q



(b) Noisy observation effects on Q_x



(c) Noisy observation effects on absolute position

Figure B.3: QXplore trained on a 'noisy tv' variant of `SparseHalfCheetah` where one element of the observation vector is normally distributed random value whose variance increases if the cheetah moves in the negative direction. The performance of QXplore is not impacted in any way by this noise, and it trains as normal.

Table B.1: Parameters used for benchmark runs.

Default Parameters	
CEM	
iterations	4
number of samples	64
top k	6
All Networks	
neurons per layer	256
number of layers	3
non-linearities	ReLU
Optimizer	Adam
Adam momentum β_1	0.9
Adam momentum β_2	0.99
Training	
Q learning rate	0.001
batch size	128
time decay γ	0.99
target Q-function update τ	0.005
target update frequency	2
TD3 policy noise	0.2
TD3 noise clip	0.5
training steps per env timestep	1
QXplore-specific	
Q_x learning rate	0.001
Q batch data ratio	0.75
Q_x batch data ratio	0.75
β_Q (Q initial output bias)	0
RND-specific	
predictor network learning rate	0.001
Extrinsic reward weight	2
Intrinsic reward weight	1
γ_{E}	0.99
γ_{I}	0.99
DORA-specific	
ϵ	0.1
β	0.05
γ_{E}	0.99
γ_{Q}	0.99
ϵ-greedy-specific	
ϵ	0.1

References

[1] Abdolmaleki, A., Springenberg, J. T., Degrave, J., Bohez, S., Tassa, Y., Belov, D., Heess, N., & Riedmiller, M. (2018a). Relative Entropy Regularized Policy Iteration. *arXiv preprint arXiv:1812.02256*.

[2] Abdolmaleki, A., Springenberg, J. T., Tassa, Y., Munos, R., Heess, N., & Riedmiller, M. (2018b). Maximum a posteriori policy optimisation. *arXiv preprint arXiv:1806.06920*.

[3] Albarghouthi, A., Gulwani, S., & Kincaid, Z. (2013). Recursive program synthesis. In *Computer Aided Verification*.

[4] Alipov, V., Simmons-Edler, R., Putintsev, N., Kalinin, P., & Vetrov, D. (2021). Towards practical credit assignment for deep reinforcement learning. *arXiv preprint arXiv:2106.04499*.

[5] Alur, R., Černý, P., & Radhakrishna, A. (2015). Synthesis through unification. In D. Kroening & C. S. Păsăreanu (Eds.), *Computer Aided Verification* (pp. 163–179). Cham: Springer International Publishing.

[6] Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., & Zaremba, W. (2017). Hindsight Experience Replay. *CoRR*, abs/1707.0.

[7] Balog, M., Gaunt, A., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016). Deepcoder: Learning to write programs.

[8] Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., & Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *NIPS* (pp. 1471–1479).

[9] Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *JAIR*, 47, 253–279.

[10] Bornholt, J., Torlak, E., Grossman, D., & Ceze, L. (2016). Optimizing synthesis with metasketches. *SIGPLAN Not.*, 51(1), 775–788.

[11] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *CoRR*, abs/1606.0.

[12] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[13] Bunel, R., Desmaison, A., Kohli, P., Torr, P. H., & Kumar, M. P. (2016). Adaptive neural compilation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16 (pp. 1452–1460). USA: Curran Associates Inc.

[14] Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2019). Exploration by random network distillation. In *ICLR*.

[15] Cai, L. X., Pizano, K., Gundersen, G. W., Hayes, C. L., Fleming, W. T., Holt, S., Cox, J. M., & Witten, I. B. (2020). Distinct signals in medial and lateral vta dopamine neurons modulate fear extinction at different times. *Elife*, 9, e54936.

[16] Castro, P. S., Moitra, S., Gelada, C., Kumar, S., & Bellemare, M. G. (2018). Dopamine: A Research Framework for Deep Reinforcement Learning.

[17] Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., & Mordatch, I. (2021). Decision transformer: Reinforcement learning via sequence modeling. *arXiv preprint arXiv:2106.01345*.

[18] Colas, C., Sigaud, O., & Oudeyer, P.-Y. (2018). Gep-pg: Decoupling exploration and exploitation in deep reinforcement learning algorithms. *arXiv preprint arXiv:1802.05054*.

[19] Collins, J., McVicar, J., Wedlock, D., Brown, R., Howard, D., & Leitner, J. (2019). Benchmarking simulated robotic manipulation through a real world dataset. *IEEE Robotics and Automation Letters*, 5(1), 250–257.

[20] de la Cruz, G. V., Du, Y., & Taylor, M. E. (2018). Pre-training with Non-expert Human Demonstration for Deep Reinforcement Learning. *CoRR*, abs/1812.0.

[21] Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., rahman Mohamed, A., & Kohli, P. (2017). Robustfill: Neural program learning under noisy i/o. In *ICML*.

[22] Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. (2016). Benchmarking Deep Reinforcement Learning for Continuous Control. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research* (pp. 1329–1338). New York, New York, USA: PMLR.

[23] Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., & Clune, J. (2021). First return, then explore. *Nature*, 590(7847), 580–586.

[24] Engelhard, B., Finkelstein, J., Cox, J., Fleming, W., Jang, H. J., Ornelas, S., Koay, S. A., Thiberge, S. Y., Daw, N. D., Tank, D. W., et al. (2019). Specialized coding of sensory, motor and cognitive variables in vta dopamine neurons. *Nature*, 570(7762), 509–513.

[25] Eysenbach, B., Gupta, A., Ibarz, J., & Levine, S. (2018). Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*.

[26] Faust, A., Francis, A., & Mehta, D. (2019). Evolving rewards to automate reinforcement learning. *arXiv preprint arXiv:1905.07628*.

[27] Feng, Y., Martins, R., Van Geffen, J., Dillig, I., & Chaudhuri, S. (2017). Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017: ACM.

[28] Feser, J. K., Chaudhuri, S., & Dillig, I. (2015). Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[29] Fickinger, A., Jaques, N., Parajuli, S., Chang, M., Rhinehart, N., Berseth, G., Russell, S., & Levine, S. (2021). Explore and control with adversarial surprise. *arXiv preprint arXiv:2107.07394*.

[30] Fox, L., Choshen, L., & Loewenstein, Y. (2018). {DORA} The Explorer: Directed Outreaching Reinforcement Action-Selection. In *ICLR*.

[31] Frankle, J., Osera, P.-M., Walker, D., & Zdancewic, S. (2015). *Example-Directed Synthesis: A Type-Theoretic Interpretation (extended version)*. Technical Report MS-CIS-15-12, University of Pennsylvania.

[32] Fu, J., Co-Reyes, J., & Levine, S. (2017). Ex2: Exploration with exemplar models for deep reinforcement learning. In *NIPS*.

[33] Fujimoto, S., Hoof, H., & Meger, D. (2018a). Addressing function approximation error in actor-critic methods. In *ICML*.

[34] Fujimoto, S., van Hoof, H., & Meger, D. (2018b). Addressing Function Approximation Error in Actor-Critic Methods. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research* (pp. 1587–1596). Stockholmsmässan, Stockholm Sweden: PMLR.

[35] Gaier, A. & Ha, D. (2019). Weight agnostic neural networks. *arXiv preprint arXiv:1906.04358*.

[36] Gaunt, A., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., & Tarlow, D. (2016). Terpret: A probabilistic programming language for program induction.

[37] Gehring, C. & Precup, D. (2013). Smart Exploration in Reinforcement Learning using Absolute Temporal Difference Errors. In *AAMAS*.

[38] Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

[39] Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11: ACM.

[40] Gulwani, S., Polozov, O., & Singh, R. (2017). Program synthesis. *Foundations and TrendsÂ® in Programming Languages*, 4(1-2), 1–119.

[41] Gvero, T., Kuncak, V., Kuraj, I., & Piskac, R. (2013). Complete completion using types and weights. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[42] Haarnoja, T., Pong, V., Zhou, A., Dalal, M., Abbeel, P., & Levine, S. (2018a). Composable Deep Reinforcement Learning for Robotic Manipulation. In *2018 {IEEE} International Conference on Robotics and Automation, {ICRA} 2018, Brisbane, Australia, May 21-25, 2018* (pp. 6244–6251).: IEEE.

[43] Haarnoja, T., Tang, H., Abbeel, P., & Levine, S. (2017). Reinforcement Learning with Deep Energy-Based Policies. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research* (pp. 1352–1361). International Convention Centre, Sydney, Australia: PMLR.

[44] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018b). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research* (pp. 1861–1870). Stockholmsmässan, Stockholm Sweden: PMLR.

[45] Haarnoja, T., Zhou, A., Ha, S., Tan, J., Tucker, G., & Levine, S. (2018c). Learning to Walk via Deep Reinforcement Learning. *CoRR*, abs/1812.1.

[46] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., & Levine, S. (2018d). Soft Actor-Critic Algorithms and Applications. *CoRR*, abs/1812.0.

[47] Hafner, R. & Riedmiller, M. (2011). Reinforcement learning in feedback control. *Machine Learning.*

[48] Hansen, N. (2006). The cma evolution strategy: a comparing review. *Towards a new evolutionary computation*, (pp. 75–102).

[49] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026–1034).

[50] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., & Meger, D. (2018). Deep Reinforcement Learning That Matters. In *AAAI-18 Conference on Artificial Intelligence*.

[51] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.

[52] Hong, Z., Fu, T., Shann, T., Chang, Y., & Lee, C. (2019). Adversarial exploration strategy for self-supervised imitation learning. In *CoRL*.

[53] Houthooft, R., Chen, X., Duan, Y., Schulman, J., De Turck, F., & Abbeel, P. (2016). Vime: Variational information maximizing exploration. In *NIPS*.

[54] Islam, R., Henderson, P., Gomrokchi, M., & Precup, D. (2017). Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. *CoRR*, abs/1708.0.

[55] Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., Quillen, D., Holly, E., Kalakrishnan, M., Vanhoucke, V., et al. (2018a). Scalable deep reinforcement learning for vision-based robotic manipulation. In *CoRL*.

[56] Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., Quillen, D., Holly, E., Kalakrishnan, M., Vanhoucke, V., & Levine, S. (2018b). Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation. In A. Billard, A. Dragan, J. Peters, & J. Morimoto (Eds.), *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research* (pp. 651–673).: PMLR.

[57] Katayama, S. (2012). An analytical inductive functional programming system that avoids unintended programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12: ACM.

[58] Katehakis, M. N. & Veinott Jr, A. F. (1987). The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2), 262–268.

[59] Khadka, S. & Tumer, K. (2018). Evolution-guided policy gradient in reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman,

N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 31* (pp. 1188–1200). Curran Associates, Inc.

[60] Kim, H., Kim, J., Jeong, Y., Levine, S., & Song, H. O. (2019). Emi: Exploration with mutual information. In *ICML*.

[61] Kitzelmann, E. (2010). *A Combined Analytical and Search-based Approach to the Inductive Synthesis of Functional Programs*. PhD thesis, Fakulät für Wirtschafts-und Angewandte Informatik, Universität Bamberg.

[62] Kumar, A., Zhou, A., Tucker, G., & Levine, S. (2020). Conservative q-learning for offline reinforcement learning. *arXiv preprint arXiv:2006.04779*.

[63] Le, V. & Gulwani, S. (2014). FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14: ACM.

[64] Levine, S., Kumar, A., Tucker, G., & Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*.

[65] Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., & Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5), 421–436.

[66] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning: Deep Deterministic Policy Gradients (DDPG). *ICLR*.

[67] Liu, E. Z., Raghunathan, A., Liang, P., & Finn, C. (2020). Explore then execute: Adapting without rewards via factorized meta-reinforcement learning.

[68] Lohani, S., Martig, A. K., Deisseroth, K., Witten, I. B., & Moghaddam, B. (2019). Dopamine modulation of prefrontal cortex activity is manifold and operates at multiple temporal and spatial scales. *Cell reports*, 27(1), 99–114.

[69] Matas, J., James, S., & Davison, A. J. (2018). Sim-to-Real Reinforcement Learning for Deformable Object Manipulation. In A. Billard, A. Dragan, J. Peters, & J. Morimoto (Eds.), *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research* (pp. 734–743).: PMLR.

[70] Metz, L., Maheswaranathan, N., Nixon, J., Freeman, D., & Sohl-Dickstein, J. (2019). Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning* (pp. 4556–4565).: PMLR.

[71] Miltner, A., Fisher, K., Pierce, B. C., Walker, D., & Zdancewic, S. (2018). Synthesizing bijective lenses. In *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2018.

[72] Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J. W., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Nazi, A., et al. (2021). A graph placement methodology for fast chip design. *Nature*, 594(7862), 207–212.

[73] 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., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529–33.

[74] Mnih, V., Mirza, M., Graves, A., Harley, T., Lillicrap, T. P., & Silver, D. (2016). Asynchronous Methods for Deep Reinforcement Learning arXiv : 1602 . 01783v2 [cs . LG] 16 Jun 2016. *CoRR*.

[75] Montague, P. R., Dayan, P., & Sejnowski, T. J. (1996). A framework for mesencephalic dopamine systems based on predictive hebbian learning. *Journal of neuroscience*, 16(5), 1936–1947.

[76] Osband, I., Aslanides, J., & Cassirer, A. (2018). Randomized prior functions for deep reinforcement learning. In *NIPS*.

[77] Parisotto, E., Song, F., Rae, J., Pascanu, R., Gulcehre, C., Jayakumar, S., Jaderberg, M., Kaufman, R. L., Clark, A., Noury, S., et al. (2020). Stabilizing transformers for reinforcement learning. In *International Conference on Machine Learning* (pp. 7487–7498).: PMLR.

[78] Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *ICML*.

[79] Pfenning, F. (2004). Automated theorem proving.

[80] Pong, V. H., Dalal, M., Lin, S., Nair, A., Bahl, S., & Levine, S. (2019). Skew-fit: State-covering self-supervised reinforcement learning. *arXiv preprint arXiv:1903.03698*.

[81] Pourchot & Sigaud (2019). CEM-RL: Combining evolutionary and gradient-based methods for policy search. In *International Conference on Learning Representations*.

[82] Redish, A. D. (2004). Addiction as a computational process gone awry. *Science*, 306(5703), 1944–1947.

[83] Rubinstein, R. Y. (1997). Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1), 89–112.

[84] Salimans, T., Ho, J., Chen, X., & Sutskever, I. (2017). Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *CoRR*, abs/1703.0.

[85] Santurkar, S., Tsipras, D., Ilyas, A., & Mądry, A. (2018). How does batch normalization help optimization? In *Proceedings of the 32nd international conference on neural information processing systems* (pp. 2488–2498).

[86] Schäfer, L., Christianos, F., Hanna, J., & Albrecht, S. V. (2021). Decoupling exploration and exploitation in reinforcement learning. *arXiv preprint arXiv:2107.08966*.

[87] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

[88] Schkufza, E., Sharma, R., & Aiken, A. (2013). Stochastic superoptimization. *SIGPLAN Not.*, 48(4), 305–316.

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

[90] Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015a). Trust region policy optimization. In *ICML*.

[91] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2015b). TRPO. *ICML*.

[92] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017a). Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.0.

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

[94] Seff, A., Ovadia, Y., Zhou, W., & Adams, R. P. (2020). Sketchgraphs: A large-scale dataset for modeling relational geometry in computer-aided design. *arXiv preprint arXiv:2007.08506*.

[95] Seff, A., Zhou, W., Damani, F., Doyle, A., & Adams, R. P. (2019). Discrete object generation with reversible inductive construction. *arXiv preprint arXiv:1907.08268*.

[96] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*.

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

[98] Simmons-Edler, R., Eisner, B., Mitchell, E., Seung, S., & Lee, D. (2019). Q-learning for continuous actions with cross-entropy guided policies. *arXiv preprint arXiv:1903.10605*.

[99] Singh, R. & Kohli, P. (2017). Ap: Artificial programming. In *SNAPL 2017*.

[100] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., & Saraswat, V. (2006). Combinatorial sketching for finite programs. In *ACM Sigplan Notices*, volume 41: ACM.

[101] Stadie, B. C., Levine, S., & Abbeel, P. (2015). Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models. *CoRR*, abs/1507.0.

[102] Sutton, R. & Barto, A. (1998). *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

[103] Sutton, R., McAllester, D., Singh, S. P., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems 12.*

[104] Sutton, R. S. & Barto, A. G. (2018). *Reinforcement learning: An introduction.* MIT press.

[105] Taiga, A. A., Fedus, W., Machado, M. C., Courville, A., & Bellemare, M. G. (2020). On bonus based exploration methods in the arcade learning environment. In *ICLR*.

[106] Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, X., Duan, Y., Schulman, J., DeTurck, F., & Abbeel, P. (2017). # exploration: A study of count-based exploration for deep reinforcement learning. In *NIPS*.

[107] Tang, Y., Nguyen, D., & Ha, D. (2020). Neuroevolution of self-interpretable agents. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (pp. 414–424).

[108] Thrun, S. B. & Möller, K. (1992). Active exploration in dynamic environments. In *NIPS*.

[109] Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, volume 16 (pp. 2094–2100).

[110] 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*, 575(7782), 350–354.

[111] Wang, R., Lehman, J., Clune, J., & Stanley, K. O. (2019). Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*.

[112] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016a). Dueling network architectures for deep reinforcement learning. In *ICML*.

[113] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016b). Dueling network architectures for deep reinforcement learning. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research* (pp. 1995–2003). New York, New York, USA: PMLR.

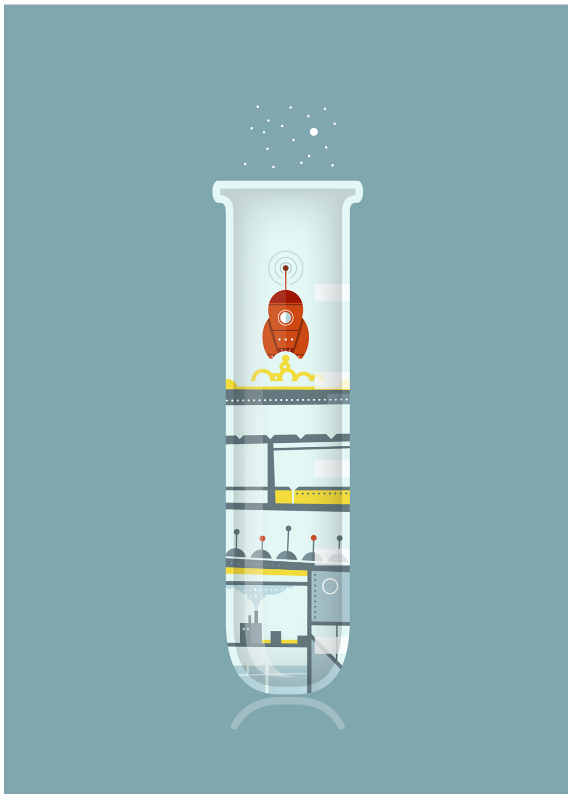
[114] Waterman, A., Lee, Y., Patterson, D. A., & Asanović, K. (2014). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0.* Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley.

[115] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning* (pp. 5–32). Springer.

[116] Wirth, C., Fürnkranz, J., & Neumann, G. (2016). Model-free preference-based reinforcement learning. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[117] Zhu, Y., Wang, Z., Merel, J., Rusu, A., Erez, T., Cabi, S., Tunyasuvunakool, S., Kramár, J., Hadsell, R., de Freitas, N., & Heess, N. (2018). Reinforcement and Imitation Learning for Diverse Visuomotor Skills. In *Proceedings of Robotics: Science and Systems* Pittsburgh, Pennsylvania.

[118] Zintgraf, L. M., Feng, L., Lu, C., Igl, M., Hartikainen, K., Hofmann, K., & Whiteson, S. (2021). Exploration in approximate hyper-state space for meta reinforcement learning. In *International Conference on Machine Learning* (pp. 12991–13001).: PMLR.



This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, "Science Experiment 02", was created by Ben Schlitter and released under cc by-nc-nd 3.0. A template that can be used to format a PhD thesis with this look and feel has been released under the permissive mit (x11) license, and can be found online at github.com/suchow/Dissertate or from its author, Jordan Suchow, at suchow@post.harvard.edu.