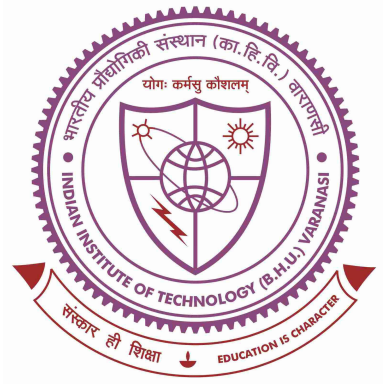
Coordinated Exploration for Concurrent Reinforcement Learning

A Dissertation
Submitted in Partial Fulfillment of the Requirements
For the Degree of
Master of Technology
in
Computer Science and Engineering

Submitted by
Shivam Garg
Roll Number: 14074017

Under the supervision of
Prof. Lakshmanan K.



Dept. of Computer Science and Engineering
Indian Institute of Technology (BHU), Varanasi
2019

Candidate's Declaration

I, Shivam Garg, certify that the work embodied in this M.Tech. thesis is my own bonafide work carried out by me under the supervision of Dr. Lakshmanan K. for a period of one year and six months from January 2018 to June 2019 at the Indian Institute of Technology (Banaras Hindu University), Varanasi. The matter embodied in this M.Tech. thesis has not been submitted for the award of any other degree or diploma.

I declare that I have faithfully acknowledged, given credit to, and referred to the research workers wherever their works have been cited in the text and the body of the thesis. I further certify that I have not willfully lifted up some other's work, paragraph, text, data, results, etc. reported in the journals, books, magazines, reports, dissertations, theses, etc., or available at websites and included them in this M.Tech. thesis and cited them as my own work.

Date: **(Shivam Garg)**
Place: **14074017**

Certificate from Supervisor/Co-Supervisor

This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.

(Dr. Lakshmanan K.) **Head of Department**
Supervisor
Assistant Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology (BHU), Varanasi

Date:

Abstract

Reinforcement Learning (RL) is concerned with the general problem of goal based decision making. The present work deals with a particular form of RL, namely Concurrent Reinforcement Learning. In Concurrent RL, multiple agents have independent access to an identical environment, with the goal being to learn to make optimal decision making in an efficient manner such that the experience of all the agents is utilized. This property is known as coordinated exploration. In this work we build upon a recently proposed method of Seed–Sampling which is based on Coordinated Exploration for Concurrent RL. We propose two types of Concurrent RL algorithms based on seed sampling: (1) A model–free policy gradient algorithm; and (2) A model–based curiosity driven exploration model. Both the methods are in the continuous state space. We also provide experimentation into how seed sampling affects exploration in an environment and its behavior when used in conjuction with varying number of parallel agents.

Acknowledgement

I would like to thank my supervisor Prof. Lakshmanan K. for providing me the platform to study this subject. He gave me directions and a stimulating environment to come up with novel ideas. His advice and guidance was invaluable to me.

I would also like to thank the thesis committee members for their time and input towards this work, and our Head of Department, Prof. Rajeev Srivastava who is most helpful and pivotal in fulfilling the academic needs of the members of our department.

The support of my family and friends was crucial during this work. They took out time from their busy lives and provided me with motivation, and also practical suggestions for this project. I am indebted to all of them.

Contents

List of Figures

1 Introduction

Reinforcement Learning (RL) refers to the broad range of algorithms whose aim is to learn goal directed optimum decision making for a particular problem. In RL formulation there is an agent and an environment. The agent interacts with the environment by performing some actions, to which the environment responds by providing the agent with a reward. Each action takes the environment from the current state to a new state. The goal is to maximize the net expected reward that the agent accumulates over an episode by taking optimal actions according to the observed state. This model is shown in Figure 1.1.

While solving problems using RL, the environment is modeled as a Markov Decision Process (MDP). An MDP $\mathcal{M}$ is a 5–tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$ which specifies the properties and dynamics of the environment. All the observable parameters of the environment which are relevant for decision making are stored in a state vector $S \in \mathcal{S}$. The actions that an agent is permitted to take form the set $\mathcal{A}$. $\mathcal{R}$ is set of scalar rewards that the environment provides the agent in a particular state. $\mathcal{P}$ is the set of transition probabilities from one state to another state upon taking a given action. MDPs follow the Markov property: the current environment dynamics depends only on the current state, and not on how that current state was reached or the sequence of past actions, observations, or rewards observed. Mathematically, it can be represented by $\Pr\{\cdot|S_0, S_1, \ldots, S_t\} = \Pr\{\cdot|S_t\}$. γ is the discount factor used for discounting of the rewards so that rewards achieved in present become more important than rewards achieved in future.

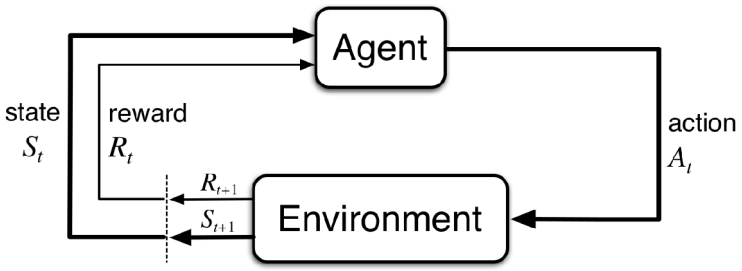


Figure 1.1: Standard RL model [1]. The environment is modeled as an MDP, which transforms from state S_t to state S_{t+1}, when the agent takes the action A_t. The environment also gives a reward R_t corresponding to the current state, which works as an evaluative feedback about the actions taken by the agent.

1.1 An example environment: CartPole

We further elaborate the RL setting by taking a concrete example of an environment: the CartPole. Fig. 1.2 shows the schematic diagram of the cart pole. In this environment, there is a pole attached by a pivot to a movable cart. The pole is free to rotate about this pivot and the cart is free to move towards left or right. We consider motions only in a 2D plane.

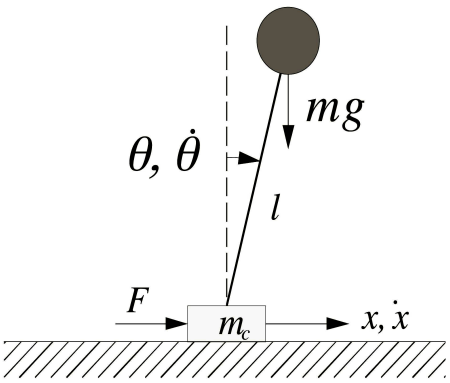


Figure 1.2: Schematic diagram of the CartPole environment [1]. The dynamics of the environment is shown for understanding purpose; an RL agent will typically not have access to this dynamics. It will either need to learn this dynamics on its own (model based learning) or work without an explicit knowledge of this dynamics (model free learning). (Figure taken from [2])

The goal of this environment is to balance the pole by physically applying a force towards left or right on the cart. If the pole drops below a certain angle or the cart moves out of a certain range on the horizontal, the episode ends. An RL agent here could be a controller installed in the cart trying to keep the pole upright. This problem can be modeled as an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, and then solved by RL methods:

State Space $\mathcal{S}$ is composed of the complete description of the cart (angle and angular velocity) and the pole (location on the horizontal and horizontal velocity): $(\theta, \dot{\theta}, x, \dot{x})$. This description serves as the observational input for the agent to act upon. The individual variables themselves have their range, for example we could model $\theta \in [-\pi/3, +\pi/3]$, ending the episode if the cart falls outside this range. Problems can have both continuous and discrete state spaces. CartPole in this formulation is a continuous state space environment.

Action Space $\mathcal{A}$ consists of all the valid actions that the agent can apply to CartPole. There are multiple ways to model it. A discrete action space could be modeled as $\mathcal{A} = \{-1, +1\}$, i.e. at each time step the agent could either push the cart left or right with some pre–determined force, say 1 Newton. A continous action space could be $\mathcal{A} = [-1, +1]$, i.e. the action can be a continuous value. In our experiments we model the CartPole action space as a discrete space.

Rewards $\mathcal{R}$ need to be modeled in such a manner that they provide evaluative feedback to the agent so that it accomplishes its goal. In CartPole, our goal is to balance the pole. So one way to model rewards is to provide a reward of $+1$ at every timestep the pole and cart are within acceptable angles and horizontal positions respectively (we use this reward formulation in our experiments). Another way could be to provide a reward of -1 when the pole falls or the cart runs away, and give a reward

of 0 otherwise. In either case, the agent will try to maximize the net accumulated reward will and learn to prevent the pole from falling down, thereby learning to keep it upright.

Transition Dynamics $\mathcal{P}$ can be composed by mathematical modeling of the physics involved in the problem, as shown in Fig. 1.2. The transition probability is of the form $P(s', a, s) = \Pr\{S_{t+1} = s' | S_t = s, A_t = a\}$, i.e. given the current state s and the action taken as a, $P(s', a, s)$ gives the probability of the next state being s'. Typically, the transition dynamics of an environment is not known to the agent a priori. The agent either estimates its own model by interacting with the environment (this is known as model based RL) or learns to perform actions without an explicit model (this is known as model free RL).

Discount Factor γ is used to weight rewards based on the timestep at which they are achieved: $\gamma^{t-1} r_t$. γ lies between 0 to 1. Discount factors are useful when we want to specify that the rewards attained at a later timestep are less important than rewards obtained at an earlier timestep. $\gamma < 1$ is also useful in limiting the net cumulative reward in case of infinite horizon environments (where the episode never ends, or is spread over a large number of timesteps).

In our experiments, we use the CartPole simulation provided by OpenAI Gym environment [4].

1.2 Policy π

An RL agent takes the state of the system as its input and outputs the optimal action, so as to maximize the cumulative reward acquired by the agent over an episode. The agent would ideally be also balancing exploration of the environment with the exploitation of known information. The function computing optimal actions given the state of the environment is known as policy. The policy is modeled as a distibution over actions:

$$\pi(a|s) = \Pr\{a_t = a | S_t = s\}.$$

The action can either be sampled from the policy distribution ($A_t \sim \pi(\cdot|S_t)$ for a stochastic policy) or the action can be chosen as the one with the maximum probability ($A_t = \max_{a \in \mathcal{A}} \pi(a|S_t)$ for a greedy policy). Thus, the goal of RL is to find a policy π for a given environment which results in the maximum possible cumulative reward being collected by the agent.

1.3 State Value function V_π

The agents maintain information about the cumulative reward achieved by following a policy π from different states using value functions. The state value function $V_\pi(s)$ is the expectation of the sum of (discounted) rewards that the agent accumulates if it starts from the state s and follows policy π (takes actions sampled from policy π), i.e.

$$V_\pi(s) := \mathbb{E}_{a\sim\pi}\Big[\sum_{i=t}^{H} R_i \Big| S_t = s\Big],$$

where H is the horizon (maximum allowable timestep of the MDP). If the agent has access to value functions, it can evaluate the "goodness" of a state, and then take actions which take it to states having higher value estimates. This way the agent can learn to act in a manner which fulfills its goal of collecting maximum cumulative reward.

1.3.1 Learning Value function in an on–policy manner

Since, the value functions are not available to the agent, they need to be learned by interacting with the environment (as shown in Fig. 1.1). This interaction results in a trajectory (list of states, actions, and rewards) $\tau^\pi := (S_0, A_0, R_1, S_1, A_1, \ldots, R_H, S_H)$, obtained by following policy π in the MDP $\mathcal{M}$.

Since, the value function is being learned for policy π and τ^π is also collected by following policy π, this type of learning is called on–policy learning. Learning in which the data is collected by following some other policy (called the behavior policy) and the value function is learned for some other policy is known as off–policy learning. We discuss off–policy learning in Section 3.1.

Once the trajectory is sampled, we formulate a learning problem as optimization of a parameterized function which approximates the value function. A form of gradient optimization technique can then be used to learn this parameterized function. This parameterized function is called a function approximator. We can model the value function using a function approximator $v_\pi(s;\theta)$ where θ is the parameter of the model. v_π can have many forms. A linear function approximator would be $v_\pi(s;\theta) = \theta^\top \phi(s)$ where $\phi(s)$ is the feature vector representation of state s. Neural networks are also a very popular form of non–linear function approximators. When neural networks are used in RL, the setting is often termed as "Deep RL".

Monte–Carlo Learning

One way to learn the function approximator is to calculate the return, sum of (discounted) rewards at each state from the sampled trajectory, and compare that with the estimate

from the value function being learned. Using this difference we formulate a loss function. For example, a type of loss function is the *mean squared value error* $\mathcal{L}_{MC}$ [1]:

$$\begin{aligned}\mathcal{L}_{MC}(\theta) &= \mathbb{E}_{s\sim\tau^\pi}\Big[\big(G(s) - v_\pi(s;\theta)\big)^2\Big] \\ &= \frac{1}{N}\sum_{t=0}^{H}\Big[\big(\sum_{k=t}^{H}\gamma^{k-t}R_k\big) - v_\pi(s_t;\theta)\Big]^2.\end{aligned} \tag{1}$$

where N is the number of samples in the trajectory, $G(S_t) = \sum_{i=t}^{H}\gamma^{i-t}R_i$ is the return from state S_t, and τ^π is the trajectory obtained by following policy π. We can then optimize this loss to find the parameter $\theta_{optimal} = \min_\theta \mathcal{L}_{MC}(\theta)$. One popular way to do it iteratively is via gradient descent: $\theta_{i+1} = \theta_i - \alpha\nabla_\theta\mathcal{L}_{MC}(\theta)$, where α is the step size. This method is called Monte–Carlo, because it samples MDPs for the trajectories τ^π and estimates the expected value (returns) from these sampled trajectories.

Temporal Difference Learning

In temporal difference learning, instead of summing the entire sequence of rewards, we use the one–step temporal difference error to learn the value function. TD(0) error δ is defined as:

$$\delta = r + \gamma v_\pi(s';\theta) - v_\pi(s;\theta),$$

where s' is the next state after s, and r is the reward given by the enivonment. We can see that TD(0) error is the difference between $v(s;\theta)$, the value estimate of state S, and $\big(R + \gamma v(S';\theta)\big)$, the value estimate of state S at the next time step after the agent has observed the reward R and the subsequent state S'.

We now describe the gradient of the *mean squared TD error* (defined analogously to the *mean squared value error* for the Monte–Carlo):

$$\nabla_\theta\mathcal{L}_{TD}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\tau^\pi}\Big[\big(r + \gamma v_\pi(s';\theta) - v_\pi(s;\theta)\big)\cdot\nabla_\theta v_\pi(s;\theta)\Big]. \tag{2}$$

We can finally train the value function approximator using gradient descent, to minimize $\mathcal{L}_{TD}(\theta)$ which will in turn minimize δ, and learn to estimate the value function V_π. Note that the gradient in the above equation has only been taken on the second term.

1.4 State–Action Value function Q_π

After the state value function is learned, we can estimate the optimal policy by the Policy Gradient algorithm, as discussed in Section 1.5. Here, we describe another value function which is very popular in RL: the Q function. Q function can be used along with policy

gradient (e.g. ACER [5], DDPG [6]), or even independently (e.g. Q–learning [7], Deep Q Network [8]).

$Q_\pi(s, a)$ function is the state–action value function which provides the estimate of the return from a state s given that the first action taken is a and thereafter policy π is followed:

$$Q_\pi(s, a) := \mathbb{E}_\pi \Big[G_t \Big| S_t = s, A_t = a \Big].$$

Q function can be learned in a similar manner as the state value function V_π. The only difference between the two is that the function approximator for Q function would have two inputs s and a: $q_\pi(s, a; \theta)$, with θ being the learnable parameter of the function approximator. For a linear function approximator, $q_\pi(s, a; \theta) = \theta^\top \phi(s, a)$, with the feature representation $\phi(s, a)$ calculated for the pair of state and action. We can learn the Q function using Monte–Carlo return or TD error, analogously to the learning of the state value function.

1.4.1 Q–learning: Off–policy learning of Q function

Learning Q function in an off–policy manner is much easier than learning a state value function off–policy. In off–policy learning of Q function, we can learn directly the Q^* function corresponding to the optimal policy π^*, while sampling the data using the behavior policy π. We let the function approximator be $q^*(s, a; \theta)$. Off–policy learning of Q function is known as Q–learning [7], [8]. It optimizes the following loss function, using gradient descent:

$$\mathcal{L}_Q(\theta) = \mathbb{E}_{(s,a,r,s') \sim \tau^\pi} \Big[\Big(r + \gamma \max_{a' \in \mathcal{A}} q^*(s', a'; \theta) - q^*(s, a; \theta) \Big)^2 \Big].$$

1.4.2 Getting the optimal policy from Q^* function

Once, we have the learned Q^* function corresponding to the optimal policy, getting the optimal policy π^* itself is straigtforward. A greedy policy can be defined as

$$\pi^*_{greedy}(s) = \arg\max_{a \in \mathcal{A}} q^*(s, a; \theta).$$

A greedy policy exploits the presently known optimal choice. A greedy policy performs no exploration and is often not good for training, especially during the initial phase of learning an MDP. To balance exploration with exploitation, a stochastic policy should be used. One such stochastic policy is the Gibbs policy:

$$\pi^*_{gibbs}(a|s) = \frac{e^{q^*(s,a;\theta)}}{\sum_{b \in \mathcal{A}} e^{q^*(s,b;\theta)}}.$$

A stochastic policy both explores and exploits in an MDP. As the learning progresses, the policy could made more greedy. This can be done by introducing a temperature parameter in the Gibbs policy.

While obtaining the optimal policy from Q^* function, we need to take a summation or an arg max operator over the whole action space $\mathcal{A}$. This can sometimes restrict the use of Q–learning in very large or continous action spaces. Though workarounds [5], [6] exist for handling this issue, policy gradient with value function learning implicitly does not have this problem.

1.5 Policy Gradient

Policy Gradient (PG) methods [9] directly learn a mapping from the state space $\mathcal{S}$ to a probability distribution over the action space $\mathcal{A}$. PG algorithms often provide superior performance when compared with pure value function based methods like Q–learning. They can also be easily extended to continuous action–space settings. In this section we consider the on–policy learning of the policy functions. The Off–policy case is discussed in Section 3.3.

We use a function approximator $\pi_\theta(a|s)$, parameterized by θ, to estimate the policy. We now look at the form of the loss function for policy gradient. The goal of RL is to maximize the expected sum of rewards accumulated by the agent. This quantity can be written as:

$$\mathcal{J}(\theta) = \sum_{s\in\mathcal{S}} d^\mu(s) V_\pi(s), \tag{3}$$

where $d^\mu(s) := \lim_{t\to\infty} P(S_t = s|S_0, A_i \sim \mu)$ is the limiting distribution of the states under the behavior policy μ. Since, we need to maximize this measure of net reward, we find θ using gradient ascent: $\theta_{t+1} = \theta_t + \alpha\nabla_\theta\mathcal{J}(\theta_t)$, where α is the step–size. We now calculate the expression for $\nabla_\theta\mathcal{J}(\theta_t)$ (the steps in this derivation follow [10]):

$$\begin{aligned}
\nabla_\theta\mathcal{J}(\theta) &= \nabla_\theta\Big[\sum_{s\in\mathcal{S}} d^\mu(s)\sum_{a\in\mathcal{A}}\pi_\theta(a|s)Q_\pi(s,a)\Big] \\
&= \sum_{s\in\mathcal{S}} d^\mu(s)\sum_{a\in\mathcal{A}}\big[\nabla_\theta\pi_\theta(a|s)Q_\pi(s,a) + \pi_\theta(a|s)\nabla_\theta Q_\pi(s,a)\big] \\
&\approx \sum_{s\in\mathcal{S}} d^\mu(s)\sum_{a\in\mathcal{A}}\nabla_\theta\pi_\theta(a|s)Q_\pi(s,a) \qquad (4)\\
&= \sum_{s\in\mathcal{S}} d^\pi(s)\sum_{a\in\mathcal{A}}\pi_\theta(a|s)\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_\theta(a|s)}Q_\pi(s,a) \\
&= \mathbb{E}_{(s,a)\sim\tau^\pi}\big[\frac{\nabla_\theta\pi_\theta(a|s)}{\pi_\theta(a|s)}Q_\pi(s,a)\big] \\
&= \mathbb{E}_{(s,a)\sim\tau^\pi}\big[\nabla_\theta\log\pi_\theta(a|s)Q_\pi(s,a)\big], \qquad (5)
\end{aligned}$$

where τ^π is the trajectory sampled from the MDP using the behavior policy $\mu = \pi$. In the above calculation: in the first step we replace $V_\pi(s)$ by $\sum_{a\in\mathcal{A}} \pi(a|s)Q_\pi(s,a)$; in the third step we ignore the second term since it is difficult to estimate it incrementally; in the fourth step we replace μ with π since our setting is on–policy (the behavior policy and the policy being learned are the same); and in the fifth step we replace the sums with expectation over the trajectory τ^π.

1.5.1 Advantage Function A_π

$Q_\pi(s_t, a_t)$ in Eq. 5 is not known and must be estimated. One way to do this is by approximating it with the returns from that step $G_t = \sum_i R_i$ (this leads to the REINFORCE algorithm [11]). However, using raw returns alone results in a high variance. The variance in REINFORCE can be controlled by subtracting from the returns a baseline estimate of the expected sum of rewards. The simplest baseline can be a mean of sum of rewards from the trajectory. A more suitable baseline is the state value function $V_\pi(s)$. With this baseline, $Q_\pi(s_t, a_t)$ is replaced by $\sum_{i=t}^{H} R_i - V_\pi(s_t)$ in policy gradient (Eq. 5). This kind of estimate is known as the advantage function $A_\pi(s,a) := Q_\pi(s,a) - V_\pi(s)$. Note that adding any baseline, that only depends on the state s, does not introduce any bias in the estimate of $\nabla_\theta \mathcal{J}(\theta)$. The following calculation with an arbitrary baseline $v(s)$, with makes it clear:

$$
\begin{aligned}
\mathbb{E}_{(s,a)\sim\tau^\pi}\left[\nabla_\theta \log \pi(a|s) v(s)\right] &= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi(a|s) v(s) \\
&= \sum_{s\in\mathcal{S}} d^\pi(s) v(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi(a|s) \\
&= \sum_{s\in\mathcal{S}} d^\pi(s) v(s) \nabla_\theta \Big(\sum_{a\in\mathcal{A}} \pi(a|s) \Big) \\
&= \sum_{s\in\mathcal{S}} d^\pi(s) v(s) \nabla_\theta \big(1\big) \\
&= 0,
\end{aligned}
$$

which shows that subtracting a baseline from the estimate of $Q_\pi(s,a)$ in Eq. 5 does not introduce any bias in the gradient.

An advantage function $A(s)$ is a relative estimate of the net–reward which can be accumulated by an agent starting from the state s. Usually, the advantage function is approximated by another function approximator, such as a parameterized value function $v_\pi(s;\theta)$.

Advantage function A_t, can have other forms too, which work well in practice [12], [13]:

$$
A_t = \delta_t + \gamma\delta_{t+1} + \cdots + \gamma^{\Lambda-t+1}\delta_{\Lambda-1}, \tag{6}
$$

where $\delta_t := R_t + \gamma V(S_{t+1}) - V(S_t)$ is the one–step TD error, $V(s)$ is the state value function, and Λ is a pre–determined timestep smaller than the episode length.

1.5.2 Simple Policy Gradient

Following Eq. 5, we can get the loss function for training θ in policy gradient:

$$\mathcal{L}_{PG}(\theta) = -\,\mathbb{E}_{(s,a)\sim\tau^{\pi}}\left[\log \pi_\theta(a|s) \cdot A_\pi(s)\right], \tag{7}$$

where τ^π is the trajectory sampled by following policy π. The negative sign in the above expression is needed because policy gradient works by gradient ascent. We use the following form of the loss function for simple policy gradient in our experiments: $\mathcal{L}_{PG}(\theta) = -\,\mathbb{E}_{(s,a,r)\sim\tau^{\pi}}\left[\log \pi_\theta(a,s) \cdot \left(\sum_i r_i - V_\pi(s)\right)\right]$.

1.5.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) Methods [14] are a class of policy gradient algorithms which have been emperically shown to provide superior performance in large scale RL problems. In PPO, the policy function is changed in such a manner that it never, in any single step, moves very far away from its previous policy. This is achieved by optimizing the following objective function:

$$\mathcal{L}_{PPO\_CLIP}(\theta) = -\,\mathbb{E}_{\tau^{\pi}}\left[\min\begin{pmatrix} \rho_t(\theta)\cdot A_t, \\ \text{clip}\big(\rho_t(\theta), 1-\epsilon, 1+\epsilon\big)\cdot A_t \end{pmatrix}\right], \tag{8}$$

where $\rho_t(\theta) := \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio, and the clip() function limits the probability ratio $\rho_t(\theta)$ from going outside the interval $[1-\epsilon, 1+\epsilon]$. This way, whenever $A_t > 0$ and $\rho_t(\theta) > 1+\epsilon$, or $A_t < 0$ and $\rho_t(\theta) < 1-\epsilon$, the loss function's gradient goes to zero and doesn't update the policy, ensuring that the policy doesn't change much from its value in the previous iteration.

This is the primary appeal of PPO. It modifies the non–linear KL–penalty of the Trust Region Policy Optimization Algorithm (TRPO [15]) into a linearly optimizable objective, while ensuring the limited update of policy in a single learning step. When learning a policy function using TRPO, we need to calculate second–order derivatives. PPO achieves, approximately, a similar objective with linear derivatives, thereby greatly speeding up the process. In problems with multi–dimensional action space, it is suitable to clip the probability ratios dimension wise [16].

1.5.4 Exact formulation of the Policy Function Approximator

In this section, we give more details about the form of function approximators used for represetning the policy in RL.

First we consider a discrete action space $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$, where n is the number of actions in the space. The function approximator in this case can be a neural network having an n–dimensional vector as its output. Once we normalize this vector, say by applying a softmax activation, we can treat the resultant normalized vector as containing the probability scores for each of the n actions. This form of a discrete distribution is called as a Categorical distribution. The actions can then be sampled from this distributions for a stochastic policy, or the action with the maximum probablility be chosen for a greedy policy.

For continuous action space we consider the example of a Gaussian Policy $\pi(a|s) = \mathcal{N}(\mu(s,a), \sigma(s,a)\boldsymbol{I})$ [1]. Instead of a function approximator outputting an array containing the probability scores for each action, we now have two function approximators $\mu(s,a)$ and $\sigma(s,a)$ giving the mean μ and standard deviation σ of the Gaussian policy as their outputs. The size of these vectors is dependent on the dimensionality of the action space $\mathcal{A}$. Note that the vector σ needs to be strictly positive. This can be achieved by adding some activation at the end of the function approximator $\sigma(s,a)$ such as an $\exp(\cdot)$ function.

1.6 Our Work

This work is concerned with **Concurrent RL Algorithms** in which multiple agents interact with an identical environment in a coordinated manner. We give background about the problem and explain an existing solution (seed–sampling [17]) to solve concurrent–RL in Section 2 .

We then extend this solution to create two new algorithms: a policy gradient model–free algorithm (Section 3.4), and a model–based curiosity driven exploration algorithm (Section 4), both in the continuous state–space domain.

Section 5 gives the details about our experimental framework and other computational features. We present the main results and their discussion in Section 6. Finally, Section 7 concludes this work.

2 Background of Coordinated Exploration

2.1 Concurrent RL

Concurrent reinforcement learning (e.g. [18]) consists of multiple independent agents interacting with different instances of an identical environment. In this setting, actions of one agent do not affect the actions of another agent. The learning algorithm in this scenario typically has access to the all the agents. As a result, an effective algorithm would take advantage of the experience of multiple agents. For example, an algorithm could have different agents expend efforts into exploring the environment differently, so that maximal information about the environment can be gathered in the least amount of time. Some common scenarios where concurrent RL arises naturally are:

Displaying advertisements online Each webpage, running on a different browser for different users, can be viewed as an agent. This agent needs to choose from a set of advertisement to display to each user, depending on the state of the browsing session, location, etc. with the goal of maximizing some metric such as the number of user clicks, or successful customer conversion to buying a product.

Autonomous self–driving cars which interact with the identical environment (such as a group of cars being run in the same part of a particular city). Each car can be viewed as an independent agent, and each car can potentially learn from its own experience as well as the experience of other cars.

Parallelizing RL algorithms so that multiple agents can be trained separately on different processors, thereby speeding up the learning process.

2.2 Coordinated Exploration

Coordinated Exploration refers to the scenario in which all the different agents, interacting with the identical environment, make a collaborative effort into exploring the environment. Each of the different agents utilizes the information gathered by all the other agents. In this work, we'll be considering a form of coordinated exploration called seed sampling [17], [19]. The general overview of the seed sampling based coordinated exploration is shown in Fig. 2.1. There are three important properties [17], which any group of coordinating agents should satisfy in order to efficiently explore the environment:

1. **Adaptivity:** The property that each agent should continually adapt to the information gathered by all the other agents.
2. **Commitment:** That the individual agents explore the environment in a manner which is fixed to that agent over a significant time–period. This will enable the agents to learn time–delayed environment statistics.

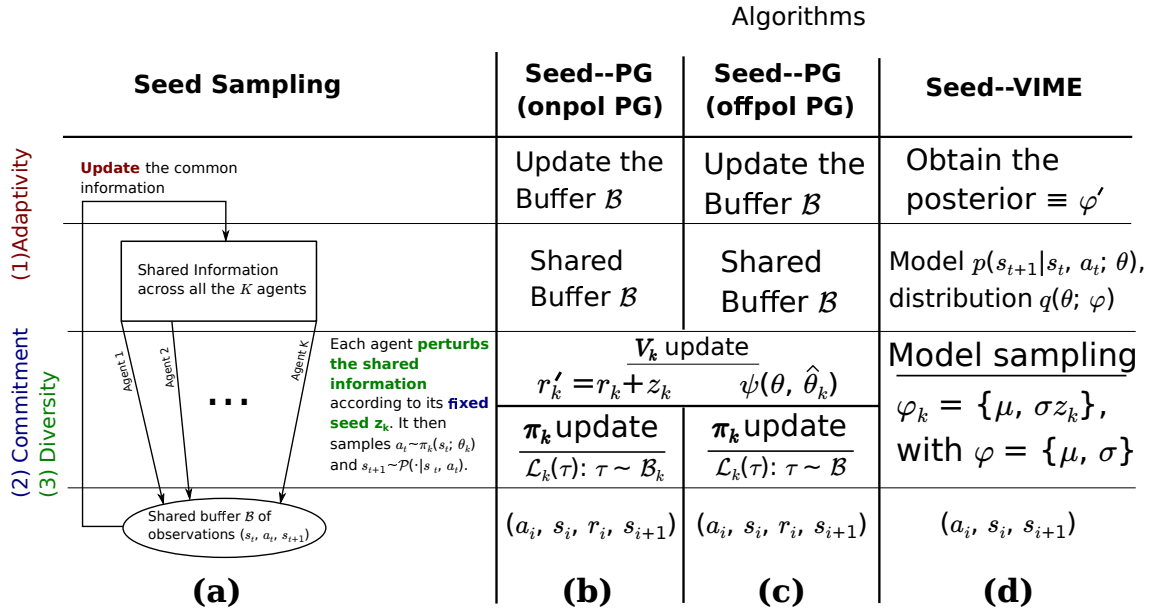


Figure 2.1: Various seed sampling algorithms. **(a)** General mechanism of seed based Coordinated Exploration; **(b)** Seed–Policy Gradient Algorithm with an off–policy learned value function and on–policy learned policy (Section 3.4.1); **(c)** Seed–Policy Gradient Algorithm with both value function and policy learned in an off–policy manner for each of the K agents (Section 3.4.2); **(d)** Seed based Variational Information Maximization Exploration for Concurrent RL (Section 4.3).

3. **Diversity:** All the agents should coordinate with each other so as to explore diversely and efficiently.

In the subsequent sections, we'll discuss two seed based schemes of coordinated exploration in concurrent RL: model based approach and model free approach.

2.3 Model–based Seed Sampling

In model–based seed sampling [17], all the agents maintain a belief over the MDP of the environment $\mathcal{M}$. Each agent then samples a seed $z_k \sim \mathcal{Z}$ at the start of the algorithm. This seed z remains fixed throughout rest of the learning. Now, each of the K agents sample an MDP $\mathcal{M}_k \sim \mathcal{M}$ through a deterministic function $\mathcal{G}$: $\mathcal{M}_k = \mathcal{G}_k(\mathcal{M}, z_k)$. The belief of the agents over $\mathcal{M}$ is periodically updated using the combined experience of all the agents.

An example of model–based seed sampling technique, which we also use in Section 4.3, is the **Standard–Gaussian Seed Sampling** [17]. In this algorithm, the agents maintain a normal prior over the reward structure of the MDP with parameters μ and Σ. Seeds are drawn as $z_k \sim \mathcal{N}(0, I)$, and the transition function $\mathcal{G}$ is defined as $\mu_t + D_t z$, where D_t is a positive definite matrix such that $D_t^\top D_t = \Sigma_t$.

2.4 Model–free Seed Sampling

The seed sampling approach to coordinated exploration discussed in the Section 2.3 can be extended to the case of continuous state space [19]. The fundamental idea of seed sampling remains the same: there is some common shared information, and each agent samples that information and perturbs it using its own value of fixed seeds. The major change from model–based seed sampling is that now the agents share, instead of a prior over the environment dynamics, a common buffer $\mathcal{B}$ which stores all the agents' observations. Further, each agent perturbs the values obtained from the common buffer, instead of sampling an MDP using its set of seeds and a deterministic function $\mathcal{G}_k$.

An example of model–free seed sampling technique is the Seed TD algorithm [19]. In seed TD, each agent maintains a different state–action value function $Q_k^\theta(s, a)$ parameterized by θ. Each agent then uses this value function to act in the environment (refer Section 1.4.2). The separate value functions for each of the agents are trained by the following algorithm:

$$\theta_n = \theta_{n-1} - \alpha \nabla_\theta \mathcal{L}_{SEED\_TD}(\theta_{n-1}),$$

with the loss function defined as

$$\mathcal{L}_{SEED\_TD_k}(\theta) = \frac{1}{N} \sum_{(s_i, a_i, r_i, s_i') \sim \mathcal{B}} \Big(r_i + z_{k,i} + \gamma \max_{b \in \mathcal{A}} Q_k^{\theta_{k,n-1}}(s_i', b) - Q_k^{\theta_{k,n-1}}(s_i, a_i) \Big)^2 + \lambda \psi(\theta, \hat{\theta}_k),$$

where ψ is a regularization penalty (e.g. $\psi(\theta, \theta_k) = \|\theta - \theta_k\|_2^2$), λ is the regularization constant, and N is the number of observations (s_i, a_i, r_i, s_i') sampled from the buffer $\mathcal{B}$. The random seed for the each agent is $\omega_k = (\hat{\phi}_k, z_{k,1}, z_{k,2}, \ldots, z_{k,H})$, and k^{th} value function Q_k^θ is initialized with the parameter $\theta_{k,0} = \hat{\theta}_k$. The actions are taken according to the policy obtained from the learned state–action value function (refer Section 1.4.2).

Note that, in this algorithm the scalar seeds z_k can be drawn from any distribution (such as the normal distribution $\mathcal{N}(0, 0.01)$), and that they remain fixed for each agent during the entirety of its training. This is necessary in order to satisfy the "Commitment" property of Coordinated Exploration (refer Section 2.2).

3 Model–free Coordinated Exploration for Policy Gradient

We present the Model–free coordinated exploration methods for policy gradient in Sections 3.4.1 and 3.4.2. But before presenting the algorithm, we first review the off–policy training of value functions in Section 3.1, and off–policy training of the policy in Section 3.3.

3.1 Learning Value Functions in an Off–Policy Manner

3.1.1 Importance Sampling

The inherent structure of the concurrent reinforcement learning requires off–policy learning of state–value function and policy function. This happens because the behavior policy μ used to generate the episodes present in the buffer $\mathcal{B}$, is different from the policy π of the agent learning from that buffer. To account for this difference, importance sampling is used. Importance sampling [1], effectively weights all the rewards obtained during an episode by the ratio of behavior policy and the policy being learned. Consider the trajectory $\tau^{\mu} = (A_t, S_{t+1}, A_{t+1}, \ldots, S_T)$ obtained from the state S_t onwards, by following the behavior policy μ. The probability of observing this trajectory is:

$$\begin{aligned}
&\Pr\{A_t, S_{t+1}, A_{t+1}, \ldots, S_T | S_t, A_i \sim \mu\} \\
&= \mu(A_t|S_t)\ P(S_{t+1}|S_t, A_t)\ \times\ \mu(A_{t+1}|S_{t+1})\ P(S_{t+2}|S_{t+1}, A_{t+1})\ \times\ \cdots \\
&= \prod_{i=t}^{T-1} \mu(A_i|S_i)\ P(S_{i+1}|S_i, A_i), \qquad (9)
\end{aligned}$$

where $\mu(A_t|S_t)$ is the probability of choosing action A_t under the policy μ given the state S_t, and $P(S_{t+1}|S_t, A_t)$ represents the environment transition probabilities. The second line in the above derivation follows from the Markov property of the environment: $\Pr\{\cdot|S_0, S_1, \ldots, S_t\} = \Pr\{\cdot|S_t\}$.

Given the above derivation, the importance sampling (IS) ratio $\rho_{t:T-1}$ is defined as the ratio of the probability of observing a sampled trajectory under π to the probability of observing the same sampled trajectory under the behavior policy μ:

$$\rho_{t:T-1} := \frac{\prod_{i=t}^{T-1} \pi(A_i|S_i)\ P(S_{i+1}|S_i, A_i)}{\prod_{i=t}^{T-1} \mu(A_i|S_i)\ P(S_{i+1}|S_i, A_i)} = \frac{\prod_{i=t}^{T-1} \pi(A_i|S_i)}{\prod_{i=t}^{T-1} \mu(A_i|S_i)}. \qquad (10)$$

The above importance sampling ratio can also be written as a product of individual one timestep importance sampling ratios $\rho_{t:T-1} = \rho_t\ \rho_{t+1} \cdots \rho_{T-1}$, where $\rho_k = \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$. With these ratios, we can now estimate the value function V_π for policy π, using the experience gathered under the behavior trajectory μ.

3.1.2 Off–policy Monte–Carlo Value Learning

The value function $V_\pi(S_t)$ is learned to estimate the net expected reward acumulated from that state: $G_t := R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T$. Since the rewards are generated using the behavior policy μ, we weight the return by the IS ratio,

$$\rho_{t:T-1} G_t = \rho_{t:T-1} R_{t+1} + \rho_{t:T-1} \gamma R_{t+2} + \ldots + \rho_{t:T-1} \gamma^{T-t-1} R_T. \tag{11}$$

In our implementations, we have used a variant of this method called the Per–Decision Importance Sampling [1]. Per–Decision IS relies on the fact that $\mathbb{E}_\mu[\rho_{t:T-1} R_{t+k}] = \mathbb{E}_\mu[\rho_{t:t+k-1} R_{t+k}]$. We give the proof of this fact:

$$\begin{aligned}
\mathbb{E}_\mu\left[\rho_k R_{k+1}\right] &= \mathbb{E}_\mu\left[\frac{\pi(A_k|S_k)}{\mu(A_k|S_k)} R_{k+1}\right] \\
&= \sum_{a \in \mathcal{A}} \mu(a|S_k) \frac{\pi(a|S_k)}{\mu(a|S_k)} R_{k+1} \\
&= \sum_{a \in \mathcal{A}} \pi(a|S_k) R_{k+1} \\
&= \mathbb{E}_\pi\left[R_{k+1}\right].
\end{aligned}$$

In the above equation, the expectation of the reward changes from $\mathbb{E}_\mu$ to $\mathbb{E}_\pi$. We obtain a similar result for two timestep IS ratios as well:

$$\begin{aligned}
\mathbb{E}_\mu\left[\rho_{k:k+1} R_{k+1}\right] &= \mathbb{E}_\mu\left[\frac{\pi(A_k|S_k)}{\mu(A_k|S_k)} \frac{\pi(A_{k+1}|S_{k+1})}{\mu(A_{k+1}|S_{k+1})} R_{k+1}\right] \\
&= \sum_{a_1, a_2 \in \mathcal{A}} \mu(a_1|S_k) \mu(a_2|S_{k+1}) \frac{\pi(a_1|S_k)}{\mu(a_1|S_k)} \frac{\pi(a_2|S_{k+1})}{\mu(a_2|S_{k+1})} R_{k+1} \\
&= \sum_{a_1, a_2 \in \mathcal{A}} \pi(a_1|S_k) \pi(a_2|S_{k+1}) R_{k+1} \\
&= \mathbb{E}_\pi\left[R_{k+1}\right]
\end{aligned}$$

Continuing in a similar manner, we obtain $\mathbb{E}_\mu[\rho_{t:T-1} R_{t+k}] = \mathbb{E}_\mu[\rho_{t:t+k-1} R_{t+k}]$, i.e. the importance sampling ratios after time step $t+k-1$ are not needed to weight the reward R_{t+k}. We can now modify Eq. 11, by replacing each $\rho_{t:T-1} R_{t+k}$ by $\rho_{t:t+k-1} R_{t+k}$ to obtain

per–decision importance sampled return $\tilde{G}_t$:

$$\tilde{G}_t = \rho_{t:t} R_{t+1} + \gamma \rho_{t:t+1} R_{t+2} + \ldots + \rho_{t:T-1} \gamma^{T-t-1} R_T, \tag{12}$$

where $\mathbb{E}_\mu[G_t] = \mathbb{E}_\mu[\tilde{G}_t]$. We approximate V_π with a neural network as a function approximator, and learn $V_\pi(S_t)$ to estimate $\tilde{G}_t$, using a loss function similar to $\mathcal{L}_{MC}$ given in Eq. 1:

$$\mathcal{L}_{MC\_IS}(\theta) = \mathbb{E}_{\tau^\mu}\left[\left(\tilde{G}_t - v_\pi(s_t;\theta)\right)^2\right], \tag{13}$$

where τ^μ is the trajectory sampled using the policy μ. The reason for using per–decision importance sampled return is that it is found to exhibit lower variance in state values as compared to the usual variant [1]. Further, by reducing the number of IS ratios (many of which would be less than 1) being multiplied, the algorithm becomes more computationally robust.

3.1.3 Off–policy Temporal Difference Based Method

Temporal difference methods to learn state value function V_π also need to incorporate importance sampling to account for the fact that the behavior policy μ is different from the target policy π. We incorporate IS ratio with the TD error $\delta_t = R_{t+1} + \gamma V_\pi(S_{t+1}) - V_\pi(S_t)$ to get a gradient of a new value function loss (analogous to $\nabla_\theta \mathcal{L}_{TD}(\theta)$ of Eq. 2) as:

$$\nabla_\theta \mathcal{L}_{TD\_IS}(\theta) = \mathbb{E}_{\tau^\mu}\left[\rho_t \delta_t \nabla_\theta v_\pi(s_t;\theta)\right], \tag{14}$$

where θ represents the weight parameters of the neural network representing the value function and τ^μ is the trajectory sampled using the policy μ. Also, again note that the gradient in the above equation has only been taken on the second term. Further, expanding all the terms in the above equation, we get the following equivalent form:

$$\nabla_\theta \mathcal{L}_{TD\_IS}(\theta) = \mathbb{E}_{(S_t,A_t,R_{t+1},S_{t+1})\sim\tau^\mu}\left[\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}\Big(R_{t+1} + \gamma v_\pi(S_{t+1};\theta) - v_\pi(S_t;\theta)\Big)\nabla_\theta v_\pi(S_t;\theta)\right].$$

3.2 Normalizing the returns: Pop–Art

The function approximator used for estimating value function in Reinforcement Learning, especially in a Monte–Carlo setting, needs to mimic the returns $G_t = \sum_i R_{t+i}$. Since the return is a cumulative sum of rewards uptill the end of an episode, they can vary accross a large range of magnitudes: a large magnitude (typically, for states in the beginning of the episode), and a small magnitude (typically, for states at the end of an episode). General function approximators are not invariant to the range of input data. This problem is usually solved by normalizing the data to have zero mean and unit variance.

However, in a reinforcement learning setting, the scale of data cannot be known a priori. This creates a need for estimating the mean and variance of the data on the fly, and to use these estimates to normalize the data. Further, the weights of the function approximators need to be suitably modified as the mean and variance estimates change in order to keep the function approximator's estimates of earlier data unchanged. In our experiments, we use a recently proposed data normalization algorithm suitable for learning values across many orders of magnitudes: Pop–Art [20]. Pop–Art algorithm is composed of two steps: (POP) Preserving Outputs Precisely and (ART) Adaptively Rescaling Targets.

(POP) Preserving Outputs Precisely works by modifying the weights of the function approximators as the mean and variance estimates from the streaming data change. This is achieved by modeling the function approximator as a linear transform on the parameterized function: $\boldsymbol{W} g_\theta(x) + \boldsymbol{b}$, where g_θ is the parameterized function, θ is the model's paramater, x is the input to the model (observation s in RL), and $\boldsymbol{W}, \boldsymbol{b}$ are the additional weights linearly transforming the parameterized function. With this model, the POP property is obtained by modifying $\boldsymbol{W}$ and $\boldsymbol{b}$ with the new mean and variance estimates, as shown in Algorithm 1.

(ART) Adaptively Rescaling Targets works by obtaining new estimates of mean and variance by weighting the earlier estimates with the mean and variance of the new batch of data just observed. The mathematical description is given in Algorithm 1.

Algorithm 1 Pop–Art Algorithm [20] for normalizing the returns.

Initialize $\boldsymbol{W} = \boldsymbol{I}, \boldsymbol{b} = \boldsymbol{0}, \sigma = 1, \mu = 0$, and finetune hyper–parameter β
while learning **do**
  Observe a batch of data: input X and output Y of size N
  *# **(ART) Adaptively rescaling targets***
  $\mu_t = (1-\beta)\mu_{t-1} + \beta \frac{1}{N} \sum_i Y_i$
  $\sigma_t = \sqrt{\nu_t - \mu_t^2}, \quad$ where $\nu_t = (1-\beta)\mu_{t-1} + \beta \frac{1}{N} \sum_i Y_i^2$
  *# **(POP) Preserving outputs precisely***
  $\boldsymbol{W} \leftarrow \sigma_{new}^{-1} \sigma \boldsymbol{W}, \quad \boldsymbol{b} \leftarrow \sigma_{new}^{-1}(\sigma \boldsymbol{b} + \mu - \mu_{new})$
  $\sigma \leftarrow \sigma_{new}, \quad \mu \leftarrow \mu_{new}$
  $\hat{Y} = \boldsymbol{W} g_\theta(x) + \boldsymbol{b}$
  Loss $\mathcal{J} = \mathcal{L}(\hat{Y}, Y)$, where $\mathcal{L}(a, b)$ is a loss function such as $\frac{1}{2}\|a-b\|_2^2$
  *# **Learning Step***
  Update $\theta, \boldsymbol{W}$, and $\boldsymbol{b}$ using the gradients $\nabla_\theta \mathcal{J}, \nabla_{\boldsymbol{W}} \mathcal{J}$, and $\nabla_{\boldsymbol{b}} \mathcal{J}$

3.3 Learning Policy Functions in an Off–Policy Manner

Once, we obtain the state value function estimate (and thereby an advantage function) via off–policy learning, we can train a policy using off–policy policy gradient. In the off–policy setting, we assume that a behavior policy μ is used to interact with the MDP, while we try to learn another policy π. We'll now review two off–policy policy gradient algorithms, both using importance sampling to accomodate the policy difference.

3.3.1 Off–Policy Actor Critic Algorithm

Off–Policy Actor Critic Algorithm (Off–PAC) algorithm [10] is a policy gradient algorithm with both the value function (also called as the critic) and the policy learned from off–policy data. Off–PAC uses importance sampling to weight the off–policy data. Continuing the derivation done in Section 1.5, we have from Eq. 4:

$$\begin{aligned}\nabla_\theta \mathcal{J}(\theta) &= \sum_{s\in\mathcal{S}} d^\mu(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q_\pi(s,a) \\ &= \sum_{s\in\mathcal{S}} d^\mu(s) \sum_{a\in\mathcal{A}} \mu(a|s) \frac{\pi_\theta(a|s)}{\mu(a|s)} \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} Q_\pi(s,a) \\ &= \mathbb{E}_{(s,a)\sim\tau^\mu} \left[\rho \nabla_\theta \log \pi_\theta(a|s) Q_\pi(s,a)\right],\end{aligned}$$

where τ^μ is a trajectory generated by following policy μ, $\rho := \frac{\pi_\theta(a|s)}{\mu(a|s)}$ is the importance sampling ratio, θ represents the weight parameters of the neural network representing the policy function. We can now give the final form of the loss function:

$$\mathcal{L}_{OFF\_PAC}(\theta) = -\,\mathbb{E}_{(s,a)\sim\tau^\mu} \left[\frac{\pi_\theta(a|s)}{\mu(a|s)} \log \pi_\theta(a|s) \cdot A_\pi(s)\right]. \tag{15}$$

Note that in Eq. 15 the policy being learned is π_θ, whereas the experience being used is generated using (possibly) another policy μ. This is off–policy actor critic algorithm. As discussed in Section 1.5, $Q_\pi(s,a)$ can be replaced by the advantage function A_π, which reduces the variance of the gradient estimates without introducing bias. We have already discussed how to estimate V_π off–policy in the previous section. Obtaining A_π is a matter of subtracting the state value function from the returns.

The Off–PAC algorithm gives an unbiased estimate of the loss function for training the policy π from data generated off–policy. However, in practice it is very difficult to use. Often the importance sampling ratios become too large or too small disturbing the learning. PPO solves this problem by clipping the IS ratios to a fixed range.

3.3.2 PPO for off–policy learning

PPO loss function (Eq. 8) extends in a straightforward way to off–policy learning [16]. We just replace $\rho_t(\theta) := \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ in Eq. 8 by the importance sampling ratio $\rho = \frac{\pi(a|s)}{\mu(a|s)}$ to get the new loss function:

$$\mathcal{L}_{OFF\_PPO}(\theta) = -\,\mathbb{E}_{\tau^\mu}\left[\min\begin{pmatrix} \rho \cdot A_\pi, \\ \text{clip}\big(\rho, 1-\epsilon, 1+\epsilon\big) \cdot A_\pi \end{pmatrix}\right], \tag{16}$$

where expectation is taken over the trajectory τ^μ generated by following policy π, $\rho = \frac{\pi(a|s)}{\mu(a|s)}$ is the importance sampling ratio. Again, A_π needs to be learned in an off–policy setting for the policy π.

The clipping function in PPO performs the important task of not learning from the importance sampling ratios that are too small or too large. As a result, in learning situations where Off–PAC may fail to learn, off–policy PPO works fine. We discuss this, in more detail, in Section 6.3.1.

3.4 Seed–PG: Learning a Family of Policy Functions

We now present our model–free coordinated exploration algorithm: Seed Policy Gradient (Seed–PG). This algorithm is directly inspired by the approach given in [19]. In Seed–PG, instead of learning a family of K separate state–action value functions Q_k corresponding to the K separate agents, we learn a family of K separate policy functions. The policy function $\pi_k^{\theta_k}(s|a)$ is parameterized by θ_k for each agent. In a way similar to the Seed TD algorithm (discussed in Section 2.4), each agent uses its seed z_k to modify its own policy gradient loss function. This leads to a differently learned policies for each agent, thereby expanding the exploratory effort.

In this setting, all the agents also maintain their independent estimates $v_{\pi_k}^{\phi_k}(s)$ of the value functions V_{π_k}; ϕ_k is the parameter of the function approximator $v_{\pi_k}^{\phi_k}$. These functions are trained (off–policy) over the combined experience of all the agents. Each agent also maintains its own buffer of experience $\mathcal{B}_k$. The common buffer $\mathcal{B}$ is the combination of all these individual experience buffers. We learn the individual value function approximators independently using either Monte–Carlo method (Eq. 13) or TD method (Eq. 19).

Seed Monte–Carlo Value Loss function

The Seed Monte–Carlo loss function for each agent is defined as:

$$\mathcal{L}_{SEED\_MC_k}(\phi_k) = \mathbb{E}_{\tau^{\pi_i}\sim\mathcal{B}}\left[\left(\tilde{G}_t^i - v_{\pi_k}^{\phi_k}(s_t)\right)^2\right], \tag{17}$$

where τ^{π_i} is the i^{th} agent's trajectory obtained by following $\pi_i^{\theta_i}$, and $\tilde{G}_t^i$ is the per–decision importance sampled return (similar to Eq. 12). The expectation is taken over trajectories sampled from the common buffer $\mathcal{B}$. We calculate $\tilde{G}_t^i$ as follows:

$$\tilde{G}_t^i = \rho_{t:t}^i(r_{t+1} + z_{k,t+1}) + \gamma\rho_{t:t+1}^i(r_{t+2} + z_{k,t+2}) + \cdots + \rho_{t:T-1}^i\gamma^{T-t-1}(r_T + z_{k,T}), \tag{18}$$

where $\rho_{t:T-1}^i = \rho_t^i\ \rho_{t+1}^i \cdots \rho_{T-1}^i$, and $z_{k,j}$s are the agent's seeds. The individual one timestep IS ratio is calculated as

$$\rho_t^i = \frac{\pi_k^{\theta_k}(a_t|s_t)}{\pi_i^{\theta_i}(a_t|s_t)},$$

where i is the agent whose trajectory is sampled, and k is the agent whose value function approximator is being learned.

Gradient of the Seed TD Value Loss function

The gradient of the Seed TD loss function is defined as:

$$\nabla_{\phi_k}\mathcal{L}_{SEED\_TD_k}(\phi_k) = \mathbb{E}_{\tau^{\pi_i}\sim\mathcal{B}}\left[\rho_t^i\delta_t\nabla_\theta v_{\pi_k}^{\phi_k}(s_t)\right], \tag{19}$$

where τ^{π_i} is the i^{th} agent's trajectory, $\rho_t^i = \frac{\pi_k^{\theta_k}(a_t|s_t)}{\pi_i^{\theta_i}(a_t|s_t)}$ is the IS ratio as before, and δ_t is the modified TD(0) error:

$$\delta_t = r_{t+1} + z_{k,t+1} + v_{\pi_k}^{\phi_k}(s_{t+1}) - v_{\pi_k}^{\phi_k}(s_t).$$

Learning the Value Loss function

At instantiation, each agent k samples its set of seeds $\omega_k = (\hat{\phi}_k, \hat{\theta}_k, z_{k,1}, z_{k,2}, \ldots, z_{k,T})$. $z_{k,j}$ could be sampled from, say a normal distribution: $z_{k,j} \sim \mathcal{N}(0, \sigma)$, σ being the standard deviation of the seeds. We initialize ϕ_k as $\phi_{k,0} = \hat{\phi}_k$. With the gradients of the loss function calculated, we can train the value function approximator using some form of gradient descent:

$$\phi_{k,j+1} = \phi_{k,j} - \alpha\left(\nabla_\phi\mathcal{L}_{SEED\_V_k}(\phi_{k,j}) + \lambda\psi(\phi_{k,j}, \hat{\phi}_k)\right),$$

where α is the step size parameter, λ is the regularization constant, and ψ is the seed regularization penalty (such as the L2 penalty $\psi(a,b) = \frac{1}{2}\|a-b\|_2^2$). In place of $\mathcal{L}_{SEED\_V}$ we use either $\mathcal{L}_{SEED\_MC}$ or $\mathcal{L}_{SEED\_TD}$.

3.4.1 Seed–PG (On–policy)

Once we have the value functions $v_{\pi_k}^{\phi_k}(s)$ available, we can learn the separate policy functions $\pi_k^{\theta_k}(s|a)$ of the agents using either on–policy or off–policy methods. On–policy methods would train only on the experience gathered by that particular agent.

We learn policy function approximators of each of the agent independently using either Simple Policy Gradient (Eq. 7) or PPO (Eq. 8). The expectation in these equations is now taken over the trajectories sampled from $\mathcal{B}_k$. The loss function for On–policy Simple PG is

$$\mathcal{L}_{SEED\_SPG_k}(\theta_k) = -\,\mathbb{E}_{\tau^{\pi_k}\sim\mathcal{B}_k}\left[\log \pi_k^{\theta_k}(a_t|s_t)\cdot A_{\pi_k}(s_t)\right]. \tag{20}$$

The loss for On–policy PPO is

$$\mathcal{L}_{SEED\_ON\_PPO_k}(\theta_k) = -\,\mathbb{E}_{\tau^{\pi_k}\sim\mathcal{B}_k}\left[\min\left(\begin{array}{l}\rho_t(\theta_k)\cdot A_{\pi_k}(s_t),\\ \text{clip}\big(\rho_t(\theta_k), 1-\epsilon, 1+\epsilon\big)\cdot A_{\pi_k}(s_t)\end{array}\right)\right]. \tag{21}$$

In both these loss functions, the symbols have their meanings as defined in the previous section. Further, we calculate $\rho_t(\theta_k) := \frac{\pi_k^{\theta_k}(a_t|s_t)}{\pi_k^{\theta_{k,old}}(a_t|s_t)}$ and $A_{\pi_k}(s_t) := G_t - v_{\pi_k}^{\phi_k}(s_t)$, where $G_t = \sum_{j=t}^{T}(r_j + z_{k,j})$. Though, the advantage could be calculated in a variety of different ways as well (Section 1.5.1).

3.4.2 Seed–PG (Off–policy)

Off–policy learning of the policy function can use samples from the entire shared buffer $\mathcal{B}$, potentially greatly expediting the learning of the MDP. We can learn the policy function using either Off–Policy Actor Critic (Eq. 15) or Off–Policy PPO (Eq. 16).

Seed Off–PAC Loss function

The loss function for Off–PAC is:

$$\mathcal{L}_{SEED\_OFF\_PAC_k}(\theta_k) = -\,\mathbb{E}_{\tau^{\pi_i}\sim\mathcal{B}}\left[\rho_t^i\cdot\log \pi_k^{\theta_k}(a_t|s_t)\cdot A_{\pi_k}(s_t)\right], \tag{22}$$

where $\rho_t^i = \frac{\pi_k^{\theta_k}(a_t|s_t)}{\pi_i^{\theta_i}(a_t|s_t)}$, $A_{\pi_k}(s_t) = \tilde{G}_t^i - v_{\pi_k}^{\phi_k}(s_t)$ and $\tilde{G}_t^i$ is calculated as given in Eq. 18. The rest of the symbols have their usual meanings as described in the previous section.

Seed Off–policy PPO Loss function

The loss function corresponding to the off–policy PPO is:

$$\mathcal{L}_{SEED\_OFF\_PPO_k}(\theta_k) = -\,\mathbb{E}_{\tau^{\pi_i}\sim\mathcal{B}}\left[\min\left(\begin{array}{l}\rho_t^i \cdot A_{\pi_k}(s_t),\\ \text{clip}\big(\rho_t^i, 1-\epsilon, 1+\epsilon\big)\cdot A_{\pi_k}(s_t)\end{array}\right)\right], \tag{23}$$

where as before $\rho_t^i = \frac{\pi_k^{\theta_k}(a_t|s_t)}{\pi_i^{\theta_i}(a_t|s_t)}$ and $A_{\pi_k}(s_t) = \tilde{G}_t^i - v_{\pi_k}^{\phi_k}(s_t)$. Rest of the symbols have their usual meaning.

3.4.3 Learning the Policy function

While learning the policy function, the same set of seeds $\omega_k = (\hat{\phi}_k, \hat{\theta}_k, z_{k,1}, z_{k,2}, \ldots, z_{k,T})$ created at the instantiation are used. θ_k is initialized as $\theta_{k,0} = \hat{\theta}_k$. We train the policy functions with the help of gradients of the loss function, using a gradient descent based optimizer:

$$\theta_{k,j+1} = \theta_{k,j} - \alpha\bigg(\nabla_\theta \mathcal{L}_{SEED\_PG_k}(\theta_{k,j}) + \lambda\psi(\theta_{k,j}, \hat{\theta}_k)\bigg),$$

where α is the step size parameter, λ is the regularization constant, and ψ is the seed regularization penalty (such as the L2 penalty $\psi(a,b) = \frac{1}{2}\|a-b\|_2^2$). In place of $\mathcal{L}_{SEED\_PG}$ we use either one of $\mathcal{L}_{SEED\_SPG}$, $\mathcal{L}_{SEED\_ON\_PPO}$, $\mathcal{L}_{SEED\_OFF\_PAC}$, or $\mathcal{L}_{SEED\_OFF\_PPO}$.

4 Model based Coordinated Exploration for Continuous State-Space

This algorithm is extension of the tabular seed based coordinated exploration [17] to the continuous state space case using curiosity based exploration [21]. In this algorithm a model of the environment is constructed and used to explore efficiently. All the agents share a prior $\mathcal{F}_0$ over the environment. Each k^{th} agent samples a model from this prior with the help of its own seeds, acts according to it, and stores its experience in the shared buffer $\mathcal{B}$. After all the K agents have acted, the prior $\mathcal{F}_{t-1}$ is updated to obtain the posterior $\mathcal{F}_t$.

Such a procedure, for the single agent case and tabular RL without curiosity, is used in Thomson Sampling [22], [23]. Thomson Sampling was extended to neural networks in [3] using Variational Inference [24], [25].

4.1 Curiosity

The dynamics of the environment is modelled as $p(s_{t+1}|s_t, a_t; \theta)$ parameterized by θ. We assume a prior $p(\theta)$ with $\theta \in \Theta$, which is updated in a Bayesian manner. If we represent the history until time step t as $\xi_t = \{s_1, a_1, \ldots, s_t\}$, then curiosity based exploration aims to maximize the sum of the reductions in entropy $\sum_t (H(\Theta|\xi_t, a_t) - H(\Theta|S_{t+1}, \xi_t, a_t)) = \sum_t I(S_{t+1}; \Theta|\xi_t, a_t)$ by choosing the optimal sequence of actions $\{a_t\}$. Further,

$$I(S_{t+1}; \Theta|\xi_t, a_t) = \mathbb{E}_{s_{t+1}\sim\mathcal{P}(\cdot|\xi_t,a_t)}\left[D_{KL}\Big[p(\theta|\xi_t, a_t, s_{t+1})\|p(\theta|\xi_t)\Big]\right], \tag{24}$$

where $D_{KL}[a\|b]$ is the KL–divergence between the distributions a and b. With this knowledge, we now augment the reward function of the system by sampling $s_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t)$ and actions $a_t \sim \pi(s_t)$:

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t) + \eta D_{KL}[p(\theta|\xi_t, a_t, s_{t+1})\|p(\theta|\xi_t)], \tag{25}$$

where $\eta \in \mathbb{R}$ is a hyperparameter. In this manner, the agent is motivated to not only visit states with high values of intrinsic reward, but also explore effectively, in the sense that it is incentivized to explore states that yield the maximum gain in information about the system dynamics – i.e. visit states which have not been visited before.

4.2 Variational Inference

One of the central problem here is the calculation of posteriors of the form

$$p(\theta|\xi_t, a_t, s_{t+1}) = \frac{p(\theta|\xi_t)p(s_{t+1}|\xi_t, a_t; \theta)}{p(s_{t+1}|\xi_t, a_t)}, \tag{26}$$

the computation of which are in general intractable. To make the problem tractable, $p(\theta|\mathcal{D})$ is modeled as a Bayesian Neural Network [26], with the distibution over its weights θ given by the distribution $q(\theta;\phi)$ parameterized by ϕ; this modeling is shown in Fig. 4.1.

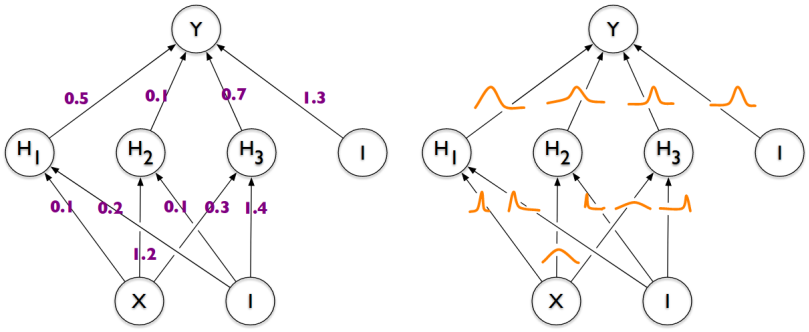


Figure 4.1: Modeling $p(\theta|\mathcal{D})$ as a Bayesian Neural Network. The neural network on the left is the classical model; in the Bayesian formulation, each of the weight parameters are assumed to come from a separate distribution as shown by the figure on the right. (Figure taken from [3]).

The distribution $q(\theta;\phi)$ is found by minimizing $D_{KL}\big[q(\theta;\phi)\|p(\theta|\mathcal{D})\big]$. Following the steps in [25], this minimization can, alternatively, be written as:

$$\begin{aligned}
\phi &= \arg\min_{\phi} \quad D_{KL}[q(\theta;\phi)\|p(\theta|\mathcal{D})] \\
&= \arg\min_{\phi} \quad \mathbb{E}[\log q(\theta;\phi)] - \mathbb{E}[\log p(\theta|\mathcal{D})] \\
&= \arg\min_{\phi} \quad \mathbb{E}[\log q(\theta;\phi)] - \mathbb{E}\left[\log\left(\frac{p(\theta)p(\mathcal{D}|\theta)}{p(\mathcal{D})}\right)\right] \\
&= \arg\min_{\phi} \quad \mathbb{E}[\log q(\theta;\phi)] - \mathbb{E}[\log p(\theta)] - \mathbb{E}[\log p(\mathcal{D}|\theta)] + \mathbb{E}[\log p(\mathcal{D})] \\
&= \arg\min_{\phi} \quad D_{KL}[q(\theta;\phi)\|p(\theta)] - \mathbb{E}[\log(p(\mathcal{D}|\theta)] \\
&= \arg\min_{\phi} \quad \mathcal{L}[q(\theta;\phi), \mathcal{D}],
\end{aligned} \tag{27}$$

where $\mathcal{L}[q(\theta;\phi), \mathcal{D}] := D_{KL}[q(\theta;\phi)\|p(\theta)] - \mathbb{E}[\log(p(\mathcal{D}|\theta)]$ is called as the *variational lower bound*, and all the expectations above are taken over θ: $\mathbb{E} \equiv \mathbb{E}_{\theta\sim q(\cdot|\phi)}$. The last term $\mathbb{E}[\log p(\mathcal{D})]$ in the fourth line above is independent of ϕ and thus can be left out. With this approximation Eq. 25 becomes

$$r'(s_t, a_t, s_{t+1}) = r(s_t, a_t) + \eta D_{KL}[q(\theta;\phi_{t+1})\|q(\theta;\phi_t)]. \tag{28}$$

Since the distribution $q(\theta;\phi)$ is implemented via a Bayesian Neural Network [26], we

formulate variational distribution q through a diagonal Gaussian distribution [3]:

$$q(\theta;\phi) = \prod_{i=1}^{|\Theta|} \mathcal{N}(\theta_i|\mu_i;\sigma^2), \tag{29}$$

where μ is the multivariate Gaussian's mean vector and σ the diagonal covariance matrix. To satisfy $\sigma > 0$ (positive definiteness) we introduce another parameter $\rho_i \in \mathbb{R}$ and let $\sigma_i = \log(1 + e^{\rho_i})$. With this formulation ϕ is defined as: $\phi := \{\mu, \rho\}$.

This method of curiosity based exploration for a single agent is called VIME (Variational Information Maximization Exploration) [21]. There are two posterior calculations in VIME. We will now discuss both of them in detail.

4.2.1 One–step posterior to calculate the information term in Eq. 25

To calculate the information term $D_{KL}[p(\theta|\xi_t, a_t, s_{t+1})\|p(\theta|\xi_t)]$ in the modified reward (Eq. 25), we need to evaluate the one–step updated posterior $p(\theta|\xi_t, a_t, s_{t+1})$ as given in Eq. 26.

If we model $p(\theta|\xi_t, a_t, s_{t+1})$ as $q(\theta;\phi')$, then using the result of Eq. 27, we get

$$\phi' = \arg\min_{\phi} \quad \Bigg[\overbrace{\underbrace{D_{KL}[q(\theta;\phi)\|q(\theta;\phi_{t-1})]}_{\ell_{KL}(q(\theta;\phi))} - \mathbb{E}_{\theta\sim q(\cdot;\phi)}[\log p(s_{t+1}|\xi_t, a_t;\theta)]}^{\ell(q(\theta;\phi),s_t)} \Bigg]. \tag{30}$$

Following the analysis in [21], we approximate the information term in the approximate modified reward function (Eq. 28), $D_{KL}[p(\theta|\xi_t, a_t, s_{t+1})\|p(\theta|\xi_t)] \approx D_{KL}[q(\theta;\phi + \lambda\Delta\phi)\|q(\theta;\phi)]$ with the update $\Delta\phi$ calculated as a single step second–order Newton update: $\Delta\phi = H^{-1}(\ell)\nabla_\phi \ell(q(\theta;\phi), s_t)$.

The fully factorized Gaussian form for q (Eq. 29 leads to a simple form for the KL Divergence $D_{KL}[q(\theta;\phi)\|q(\theta;\phi')]$ is:

$$\begin{aligned}
D_{KL}[q(\theta;\phi)\|q(\theta;\phi')] &= \mathbb{E}_{\theta\sim q(\cdot;\phi)}[\log q(\theta;\phi)] - \mathbb{E}_{\theta\sim q(\cdot;\phi)}[\log q(\theta;\phi')] \\
&= \mathbb{E}\log\prod_{i=1}^{|\Theta|}\mathcal{N}(\theta_i|\mu_i;\sigma_i) - \mathbb{E}\log\prod_{i=1}^{|\Theta|}\mathcal{N}(\theta_i|\mu_i;\sigma'_i) \\
&= \mathbb{E}\sum_{i=1}^{|\Theta|}\log\mathcal{N}(\theta_i|\mu_i;\sigma_i) - \mathbb{E}\sum_{i=1}^{|\Theta|}\log\mathcal{N}(\theta_i|\mu_i;\sigma'_i) \\
&= \sum_{i=1}^{|\Theta|}\mathbb{E}\left[\log\frac{1}{\sqrt{2\pi\sigma_i^2}}e^{-\frac{(\theta_i-\mu_i)^2}{2\sigma_i^2}} - \log\frac{1}{\sqrt{2\pi\sigma_i'^2}}e^{-\frac{(\theta_i-\mu'_i)^2}{2\sigma_i'^2}}\right]
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{|\Theta|} \left(\log \sigma_i' - \log \sigma_i - \frac{\mathbb{E}[(\theta_i - \mu_i)^2]}{2\sigma_i^2} + \frac{\mathbb{E}[\theta_i^2] + \mu_i'^2 - 2\mu_i' \mathbb{E}[\theta_i]}{2\sigma_i'^2} \right) \\
&= \frac{1}{2} \sum_{i=1}^{|\Theta|} \left(2\log \sigma_i' - 2\log \sigma_i - \frac{\sigma_i^2}{\sigma_i^2} + \frac{\mu_i^2 + \sigma_i^2 + \mu_i'^2 - 2\mu_i' \mu_i}{\sigma_i'^2} \right) \\
&= \frac{1}{2} \sum_{i=1}^{|\Theta|} \left(\left(\frac{\sigma_i}{\sigma_i'} \right)^2 + 2\log \sigma_i' - 2\log \sigma_i + \frac{(\mu_i' - \mu_i)^2}{\sigma_i'^2} \right) - \frac{|\Theta|}{2}, \qquad (31)
\end{aligned}
$$

where the expectation $\mathbb{E}_{\theta \sim q(\cdot;\phi)}$ has been replaced by $\mathbb{E}$ for brevity from second line onwards.

Further, if we approximate $H(\ell) \approx H(\ell_{KL})$, from Eq. 31 we can see that $\frac{\partial^2 \ell_{KL}}{\partial \mu_i^2} = \frac{1}{\log^2(1+e^{\rho_i})}$ and $\frac{\partial^2 \ell_{KL}}{\partial \rho_i^2} = \frac{2e^{2\phi_i}}{(1+e^{\rho_i}} \frac{1}{\log^2(1+e^{\rho_i})}$ defines the Hessian $H(\ell)$ with all the non–diagonal entries being zero.

Finally, we approximate the KL divergence through a second–order Taylor expansion:

$$
D_{KL}[q(\theta; \phi + \lambda \Delta\phi) \| q(\theta; \phi)] \approx \frac{1}{2} \lambda^2 \nabla_\phi \ell^\top H^{-1}(\ell_{KL}) \nabla_\phi \ell. \qquad (32)
$$

We can now use Eq. 32 in Eq. 28 to calculate the updated reward r' at each step of the agent's trajectory.

4.2.2 Updating the common posterior after observing a large chunk of data

The second posterior we need to calculate is the update in the model parameters themselves after, say, all the K agents have completed an episode. The combined observations from all the agents would be stored in the shared observation buffer $\mathcal{B}$. To perform this posterior update, we sample $\mathcal{D} \sim \mathcal{B}$.

We can approximate $\mathcal{L}[q(\theta; \phi), \mathcal{D}] := D_{KL}[q(\theta; \phi) \| p(\theta)] - \mathbb{E}[\log(p(\mathcal{D}|\theta)]$ by replacing the expectation with summation over θ (as given in [3]):

$$
\mathcal{L}[q(\theta; \phi), \mathcal{D}] \approx \sum_{i=1}^{n} \left(\log q(\theta_i; \phi) - \log p(\theta_i) - \log p(\mathcal{D}|\theta_i) \right), \qquad (33)
$$

where $\mathcal{D}$ would be sampled from the shared buffer $\mathcal{B}$, and n is the number of independent samples of θ considered. Further, the last term in the above equation can be caclulated as

$$
p(\mathcal{D}|\theta_i) = \prod_{(s_i, a_i, s_i') \sim \mathcal{D}} p(s_i'|s_i, a_i; \theta_i),
$$

where $p(s_i'|s_i, a_i; \theta_i)$ gives the probability of $s_{t+1} = s_i'$ given that $s_t = s_i, a_t = a_t$, and $\theta = \theta_i$.

Now with the help of Eq. 33, $q(\theta; \phi) \approx p(\theta|\mathcal{D})$ can be calculated via backpropogation and first–order optimization techniques. This procedure is known as Bayes by Backprop [3].

4.3 Seed VIME

We now present the complete VIME algorithm for concurrent RL using seed sampling. This algorithm can effectively be seen as the extension of the Standard–Gaussian Seed Sampling [17] (also discussed in Section 2.3) to the continuous state space.

In Seed–VIME we maintain a shared model $p(s_{t+1}|s_t, a_t; \theta)$ of the system dynamics, having a corresponding distribution $q(\theta, \phi)$ over the weights θ of the model p. Each agent k gets initialized with its own set of seeds z_k. At the beginning of each episode, the agents separately sample the system dynamics by perturbing the parameter ϕ of the distribution $q(\theta; \phi)$ using its own set of seeds, thereby obtaining ϕ_k. During the episode, the agent acts according to its policy π_k and generates the modified rewards for its trajectory, using the system dynamics governed by ϕ_k. At the end of the episode, the agents update their individual policy π_k using the generated sequence of modified rewards.

At the end of the episodes for all the agents, the shared model is updated using observations sampled from the shared buffer $\mathcal{B}$.

Algorithm 2 Seed–VIME for Concurrent RL

for each agent $k \in \{1, 2, \ldots, K\}$ **do**
  Generate seed $z_k = \mathcal{N}(0, I)$ # *initialization*
for each epoch n **do**
  # *clear buffer* $\mathcal{B}$
  for each agent $k \in \{1, 2, \ldots, K\}$ **concurrently do**
    Let $\phi_k = \{\mu, \sigma z_k\}$, with $\phi = \{\mu, \sigma\}$ # *sample the environment dynamics*
    for each timestep $t_k \in \{1, 2, \ldots, H\}$ **do**
      Generate action $a_t \sim \pi_k(s_t; \theta_k)$ and get the next state $s_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t)$
      Add the experience (s_t, a_t, s_{t+1}) to the shared buffer $\mathcal{B}$.
      Construct $r'(s_t, a_t, s_{t+1})$, using Section 4.2.1 with $\phi = \phi_k$
    Use the sequence of rewards $\{r'(s_t, a_t, s_{t+1}\}$ to update the policy $\pi_k(s)$
  Update ϕ by minimizing $\mathcal{L}[q(\theta; \phi), \mathcal{D}]$, using Section 4.2.2 with $\mathcal{D}$ sampled from $\mathcal{B}$

In Seed–VIME, all the different agents sample different models of the environment dynamics, and then explore according to this system dynamics in a curiosity–driven fashion. This algorithm's general framework is compared to that of other seed–sampling algorithms in Fig. 2.1.

5 Experimental Setup

5.1 Distributed Data Pipeline

The seed based coordinated exploration method consists of multiple agents interacting independently with different instances of a similar environment, in a concurrent fashion. Though, their runs are indpendent of each other, each agent shares its experience with all the other agents. The experience of each agent is composed of the complete episode trajectory $(S_0, A_0, \pi(A_0|S_0), R_1, \ldots S_0, A_T, \pi(A_T|S_T), R_T)$, where S_i represents the observation state, A_i represents the action taken by the behavior agent, $\pi(A_i|S_i)$ represents the probability of taking action A_i, and R_{i+1} represents the reward obtained from the environment, all at timestep i.

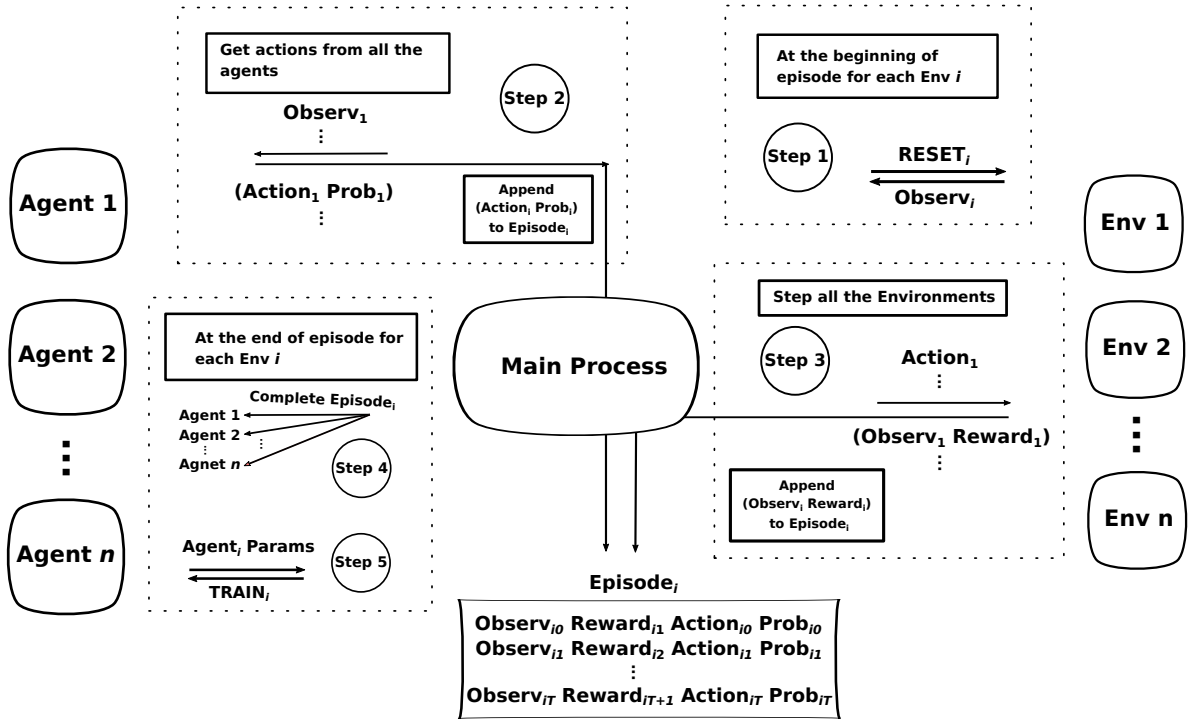


Figure 5.1: Distributed Data Pipeline: Schematic view of the data pipeline used in experimentation of seed–sampling based algorithms. Different processes are represented by rounded rectangles, and the arrows represent the exchange of data between the main process and the processes corresponding to the agents and the environments. The dotted rectangles enclose the major steps in the training procedure, as outlined in Algorithm 3.

Our implementation of seed–sampling based coordinated exploration is motivated from the implementation of BatchPPO [27]. BatchPPO is also a distributed data framework, in which a single agent interacts with a batch of environments. In RL, the data collection from environments is often a bottleneck in training the RL agent. BatchPPO alleviates this problem by running multiple environments in parallel on different processors.

Our framework, extends the functionality of BatchPPO to multiple agents as well. This way, there are multiple agents interacting with multiple environments. Further, BatchPPO was implemented using the automatic differentiation package TensorFlow [28]. We implement our framework using another package PyTorch [29]. We choose PyTorch because of its more pythonic interface and better dynamic graph support than TensorFlow. We now give additional details about our data processing pipeline.

To experiment with the seed based RL methods, we implement a distributed data pipeline framework, as shown in Fig. 5.1. In this framework, there is a *Main Process* which synchronized the data passing between different environments and agents. Each environment and agent is run on a separate process, implemented by using the `multiprocessing` library in Python language. This way, the processes bypass the Global Interpreter Lock (GIL) of Python, and are able to truly run in a parallel way on a multi–core computing device.

We outline the exact mechanism of data transfer, in a simplified way, in Algorithm 3. This algorithm in conjuction with Fig. 5.1 illustrates the working of the framework.

Algorithm 3 Main loop for the distributed data pipeline, as shown in Fig. 5.1.

while True **do**
  # ***Step 1***
  for each environment i, which is beginning its episode **do**
    Send `RESET` signal to the environment i.
    Obtain state $S_{i0} \sim \nu(s)$, where $\nu(s)$ is the initial state distribution.
  # ***Step 2***
  for each Agent i **do**
    Send S_i to the Agent i.
    if i^{th} Agent is not training **then** # *as illustrated in Fig. 5.2*
      Obtain $A_i, \pi_i(A_i|S_i) \sim \pi_i$, where π_i is the policy of the i^{th} Agent.
      Append the data $A_i, \pi_{\theta_i}(A_i|S_i)$ to the i^{th} episode.
  # ***Step 3***
  for each Environment i **do**
    Send A_i to the environment i.
    Obtain $S'_i, R_i \sim P_i(S', R|S, A)$, where P_i is the environment dynamics.
    Append the data S'_i, R_i to the i^{th} episode.
  for each environment i, which has completed its episode **do**
    # ***Step 4***
    Send the completed episode $(S_0, A_0, \pi(A_0|S_0), R_1, \ldots S_0, A_T, \pi(A_T|S_T), R_T)$ to each of the Agents.
    # ***Step 5***
    Obtain Agent parameters θ_i from the Agent for logging purposes.
    Send `TRAIN` signal to the Agent i (Section 3.4).

Further, we keep the interaction between the agents and the environments synchronized to maintain an efficient implementation. This means that all the environments are

stepped in parallel synchronously, by providing them with their corresponding actions. Then, the observation vectors and the rewards are collected from all the environments. The observation vectors are passed to the corresponding agents, and the agents are signalled to output an action synchronously. This process continues until some environment's episode ends. This is illustrated in the left part of Fig. 5.2.

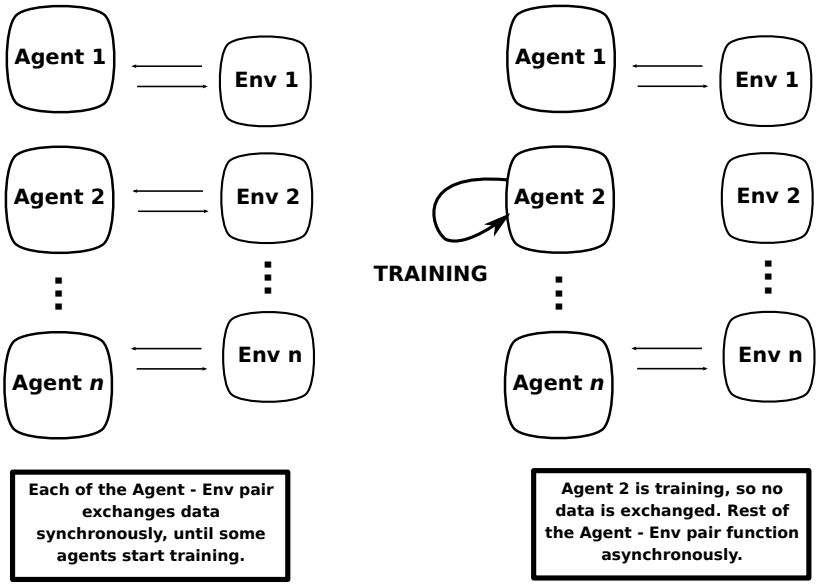


Figure 5.2: Mechanics of data exchange in the distributed data pipeline. Each of the Agent–Environment pair exchanges State, Reward, and Action in a synchronous fashion via the main process, as shown in the left figure. After the end of the episode of an agent, say Agent 2, that agent starts training and is no longer available to provide actions to the corresponding environment. At this stage, this particular Agent–Environment pair stops exchanging data, while the other pairs function in the usual manner. This is shown in the right figure.

Once, the episode of an agent completes, that particular agent starts training on its collected buffer of episodes. During the training, this agent does not perform any action. So, the data exchange between this Agent–Environment pair is halted. All the other Agent–Environment pairs continue to function as described above. The right part of Fig. 5.2 shows this asynchronous behavior of the data pipeline.

5.2 Choosing the step size α and Pop–Art weighting parameter β for value function learning

We select the optimal step size α for value function and the weighting parameter β for Pop–Art by performing a grid search over $\alpha \in \{0.0001, 0.001, 0.01, 0.1, 1.0\}$ and $\beta \in \{\text{None}, 0.1, 0.3, 0.7, 0.9, 1.0\}$. Specifically, α refers to the step–size of the Adam optimizer [30] used to train the neural network corresponding to the value function and β refers to the Pop–Art weighting parameter used to calculate running estimates of mean and

variance as given in Algorithm 1. We experiment by training a single RL agent on the CartPole environment (Section 1.1) using the Proximal Policy Optimization algorithm with a Monte–Carlo return based value function.

Fig. 5.3 shows the experimental results for the top performing pairs of (α, β). The top performing pairs, which achieve the highest cumulative reward earliest, are $\alpha = 0.001, 0.01$ for $\beta = 0.1$; $\alpha = 0.1$ for $\beta = 0.3$; and $\alpha = 0.01$ for the variant which is not using any normalization. However, "Value Loss" graph shows that the variant without any normalization suffers from high loss magnitude throughout its training, whereas the variants using Pop–Art have a significantly less loss magnitude. High loss often leads to unstability and is not preferred. The rest of the three top performing pairs exhibit comparable performance, and are able to enjoy high learning rates without the high loss seen in the variant without normalization. For our main experiments we choose $\alpha = 0.001$ and $\beta = 0.1$.

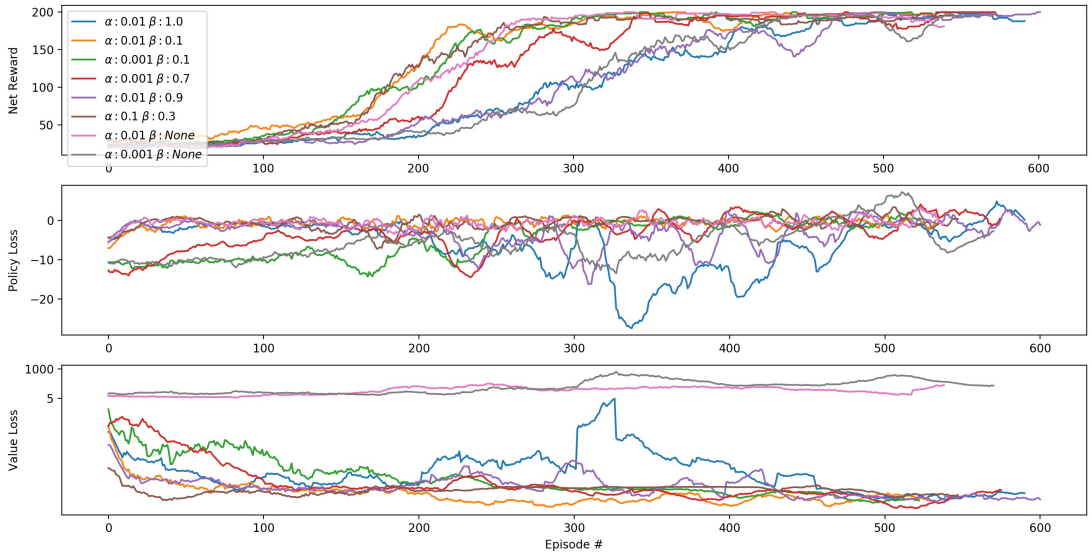


Figure 5.3: Experimental runs of a single RL agent (running PPO) on CartPole environment, with different step sizes α and Pop–Art weighting parameter β. The runs with (β : None) don't use any return normalization. We show the top performing experimental runs. The graphs are plotted by taking a moving average window of 25 over the data. The "Value Loss" subplot uses two linear scales [0–5] and [5–1000] on the Y axis.

We conducted similar experiments to tune α and β with Simple Policy Gradient algorithm. The results were very similar to the runs with PPO, so we do not show them here.

We also conducted experiments (not shown here) to tune the learning rate of the function approximator representing the policy, and found $\alpha_\pi = 0.001$ and $\alpha_\pi = 0.01$ to have the best and comparable performances. We fix the former for our experiments.

5.3 Choosing the PPO clipping factor ϵ

The clipping factor ϵ in the PPO Algorithm [14] limits the gradient from noisy importance samples. The importance ratio ρ gets clipped between $1-\epsilon$ and $1+\epsilon$. As a result, the policy does not change too much from its previous state, resulting in stable learning. Further, the gradients for updates with too large or too small importance samples vanish without affecting the policy learning. We tune ϵ by searching over $\{0.0, 0.2, 0.5, 0.7, 1.0\}$.

Fig. 5.4 shows the results of the experiments performed with 8 parallel agents, with seed samples drawn from $\mathcal{N}(0, 0.01)$. The plot shows the average performance of the 8 agents. "Deviation in ρ" graph shows the average deviation of the importance sampling ratios from 1, defined as $|\rho - 1| + 1$. "% clipped" shows the average fraction of data points neglected while training the policy, due to clipping of the IS ratio. From the graph, $\epsilon = 0.2$ achieves the best performance, and suffers from less than 10% data samples being clipped while training. Although, $\epsilon = 1.0$ considers almost all the samples while training (almost zero clipping), its performance is drastically affected negatively from the noisy IS ratios as seen from "Deviation in ρ". On the other hand $\epsilon = 0.0$ has very well behaved IS ratios, but clips about 50% of its data samples. With such a large clipping, its performance is also suboptimal to $\epsilon = 0.2$.

We choose $\epsilon = 0.2$ as the clipping factor in our experiments later.

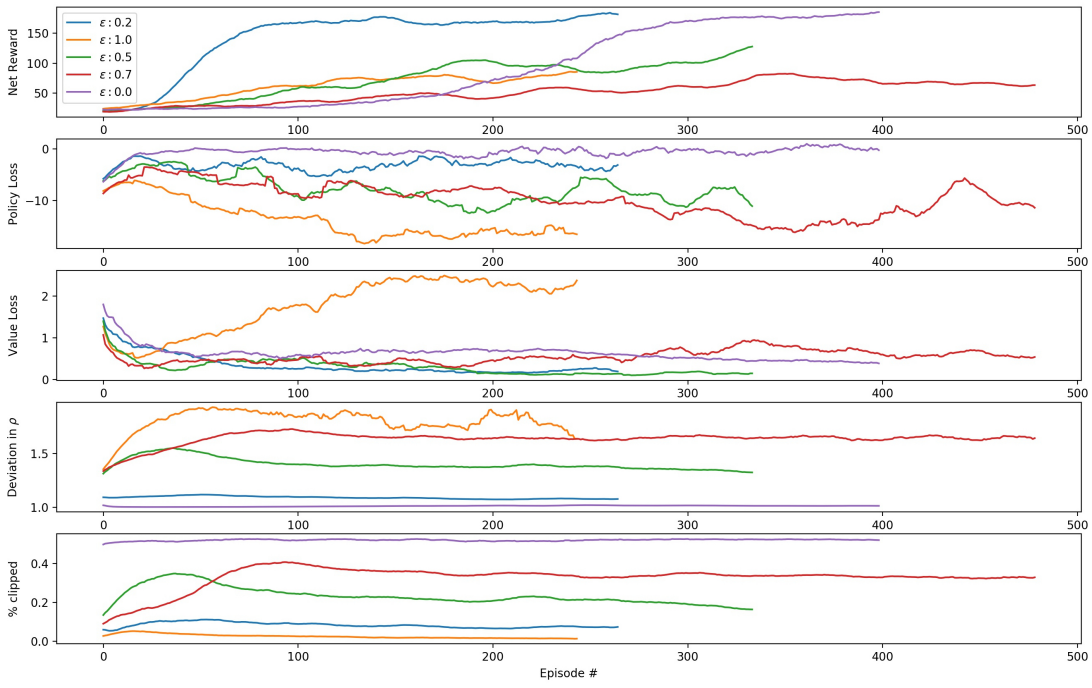


Figure 5.4: Average performance measures of 8 independent and indentical RL agent (running PPO) on the CartPole environment, with different values of the PPO clipping factor ϵ. The graphs are plotted by taking a moving average window of 25 over the data. The data itself is the mean of the values from the 8 different agents.

6 Results and Discussions

We now present our main results. All the experiments are performed using the distributed data pipeline discussed before. The seeds z_k for each agent k are sampled as $z_{k,j} \sim \mathcal{N}(0, \sigma)$. For learning the state value function, we use a per–decision IS sampling value function with Pop–Art which trains in an off–policy manner. For training the policy, we experiment with PPO and Simple Policy Gradient (both on–policy), and PPO and Off–PAC (both off–policy). Further, we use the Adam optimzer [30] during updates and do not use the seed regularization penalty described in Section 3.4.

6.1 Effect of adding seeds on state exploration

In this section we compare the effect of adding seeds of different standard deviations to rewards while learning. The seeds are sampled from a standard normal distribution $\mathcal{N}(0, \sigma)$ with mean 0 and fixed standard deviation σ. We plot the reward and loss graphs in Fig. 6.1, and visualize the states visited by the agent during two different time periods using Parallel Coordinates (refer Appendix A) in Fig. 6.2.

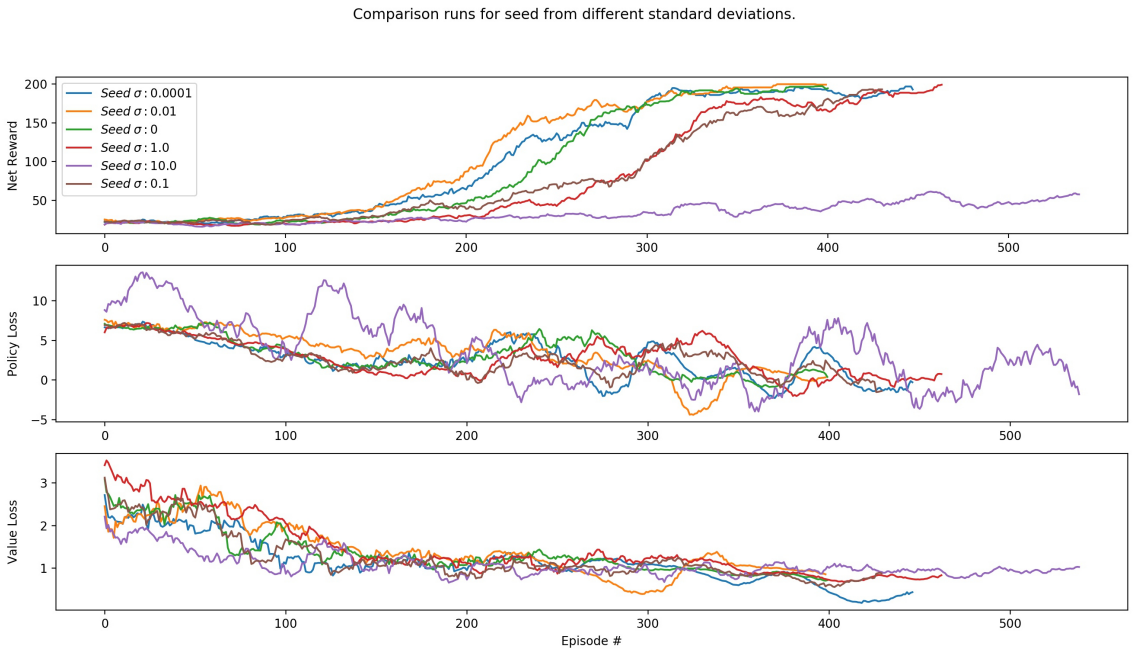


Figure 6.1: Experimental runs of a single RL agent, running simple policy gradient (Section 1.5.2) on CartPole environment, with different standard deviation values σ of the seed values. The graphs are plotted by taking a moving average window of 25 over the data.

Fig. 6.1 shows that the top three performing agent are ones with Seed $\sigma = 0.01$, followed by agent with $\sigma = 0.001$ and then the agent with no seed sampling. Further,

all the agents, except the one with Seed $\sigma = 10.0$, are able to achieve a maximum net cumulative reward of 200. We'll now connect these observations with the state visitation graphs.

State visitation graphs of the agents (Fig. 6.2) visualize the state visitation of each of the agents at the end of the 230^{th} episode (top row) and at the end of its final episode of training (bottom row). From this figure, on comparing the state visitation at the end of 230^{th} episode, of agent ($\sigma = 0.01$) with agent ($\sigma = 0.0$) we find that the former agent has explored a larger variety of state spaces. This is visible from the two left most dimensions x and v; here agent ($\sigma = 0.01$) has more coverage than all the other agents. As a result, the performance of agent ($\sigma = 0.01$) is better at the end of the 230^{th} episode than all the other agents, because it has gained more experience in terms of the variety of state spaces covered.

Another observation we can make is that in the plots at the end of the training for each agent, all the agents have explored roughly the same states, except for agent ($\sigma = 10.0$) which has the least explored state space. This is in agreement with the performance of this agent in Fig. 6.1; only this agent is unable to attain the maximum reward cumulative reward of 200.

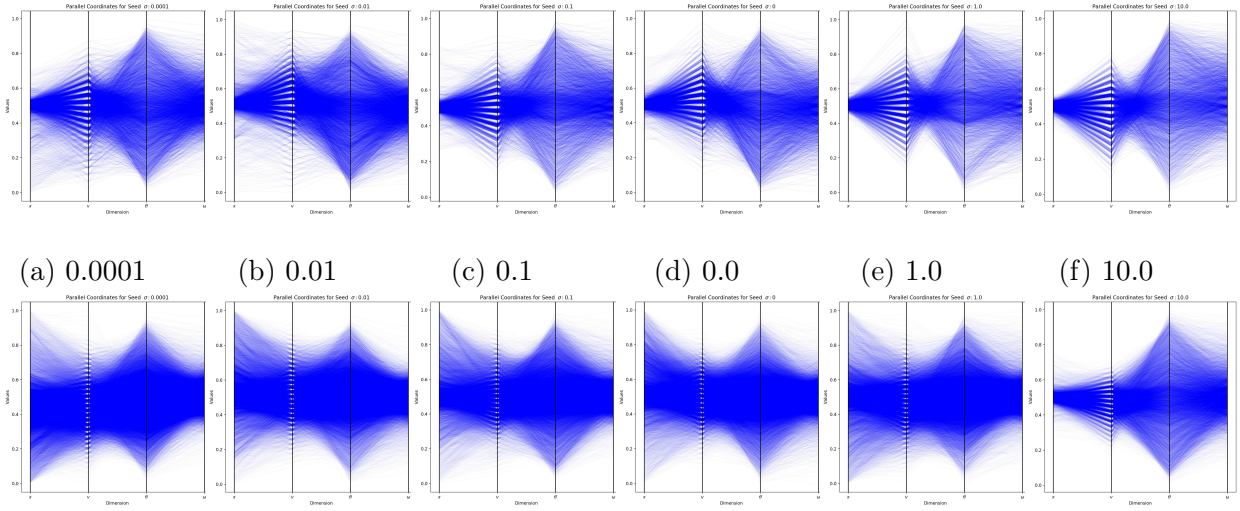
From these plots, we can infer that seed sampling encourages exploration during the initial stages of training, which leads to faster learning of the agent. Further, seeds which have high standard deviation can hamper learning by making the weight parameter updates of the agent noisy.

6.2 On–policy Seed sampling based policy gradient

In this section, we show the performance of Seed–PG (on–policy policy gradient) for multiple agents. We use simple policy gradient algorithm (Section 1.5.2) with a neural network based policy and value function. The agents share the experience buffer for off–policy value function learning (Section 3.1), but update their own policies only on their individual buffers. Fig. 6.3 shows the average performance for 4 and 16 parallel agents using different standard deviation for seeds. It also show the individual reward performance of each of the 16 agents for seeds $\sigma = 0.1$ and $\sigma = 0.0$.

We observe from the graphs that different seed values for the agents do not result in any significant improvement in the performance the agents. Although, seed $\sigma = 10.0$ decreases the performance of the agents in both the case of 4 and 16 agents. Further, comparing the Fig. 6.3a and 6.3b, we see that increasing the agents from 4 to 16 doesn't lead to any improvement in the rate of learning of the agents as well. This suggests that, not sharing the experience buffer across agents while training the policy (on–policy learning) seriously limits the rate of learning of the agents.

However, if we plot the individual agents performances of the agents separately (as



(a) 0.0001 (b) 0.01 (c) 0.1 (d) 0.0 (e) 1.0 (f) 10.0

(g) 0.0001 (h) 0.01 (i) 0.1 (j) 0.0 (k) 1.0 (l) 10.0

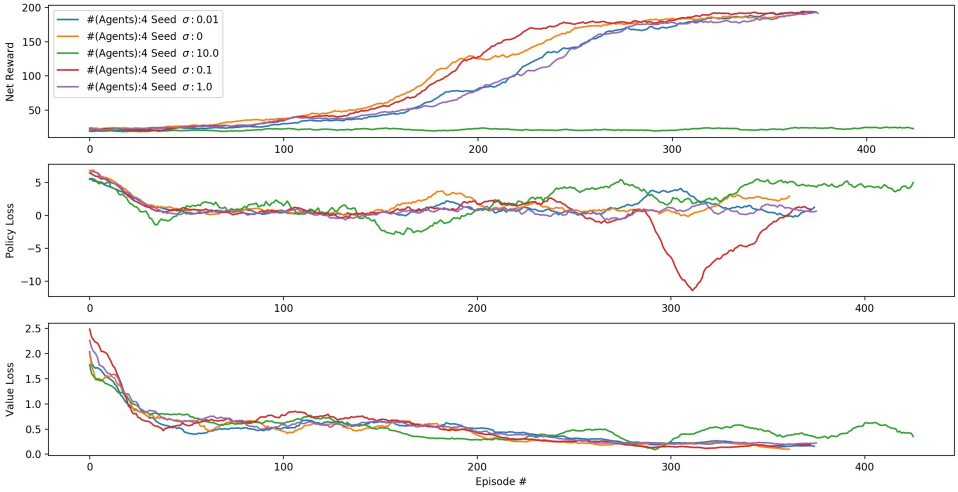
Figure 6.2: State visitation represented using parallel coordinates (refer Appendix A), for single agents (running simple policy gradient) using different seeds on CartPole. The graphs in the top row show the state visitations for the first 230 episodes experienced by the agents. The graphs on the bottom row represent state visitations for all the episodes visited by the agents during their entire course of training. The titles of the subgraph mention the value of seed σ for that plot.

done in bottom two subgraphs 6.3c and 6.3d), we observe that the many of the agents with seed $\sigma = 0.1$, are able to learn faster than many of the agents with seed $\sigma = 0.0$. Further, many agents with $\sigma = 0.1$ perform worse than agents with seed $\sigma = 0.0$, leading to similar average performance as seen in Fig. 6.3b. There exists a wider spread in the agent performances of $\sigma = 0.1$.

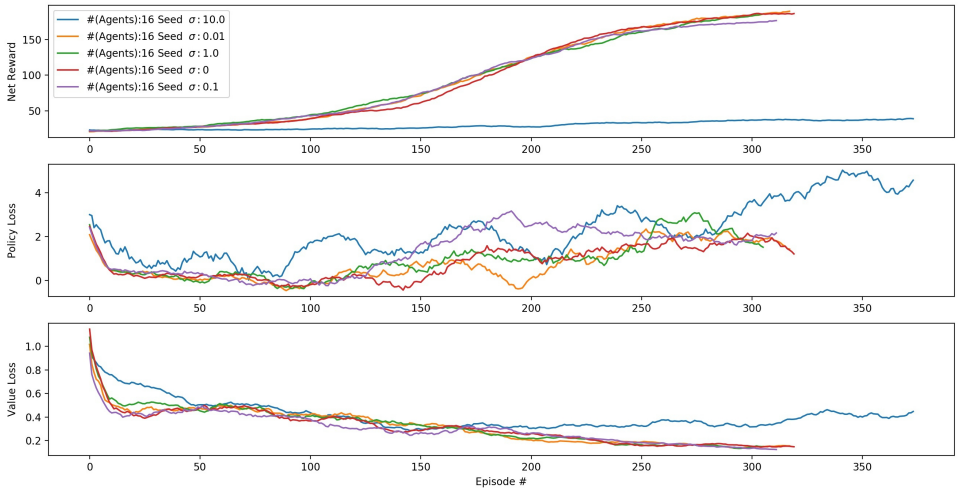
6.3 Off–policy Seed sampling based policy gradient

We now discuss the performance of Seed–PG with off–policy policy gradient. We train 4 and 16 parallel agents using PPO algorithm (Section 3.3.2), with a clipping factor $\epsilon = 0.2$. In this setup, the agents share the experience buffer for learning both the value function and the policy. As before, we use neural networks as function approximators. Figure 6.4 shows the experimental runs.

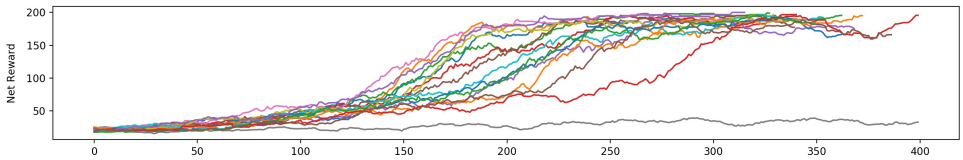
First observation we make from Fig. 6.4a, 6.4b, and 6.1 is that increasing the number of agents greatly improves the speed of learning of the MDP. A single agent (Fig. 6.1) is able to get 200 cumulative reward at about 300^{th} episode; 4 agents (Fig. 6.4a) attain a similar performance at less than 150 episodes; and 16 agents (Fig. 6.4b) attain a similar performance at about 30^{th} episode. This is in sharp contrast with the on–policy Seed–PG algorithm, where increasing the number of agents did not (or just marginally) improve the average performance of the agents.



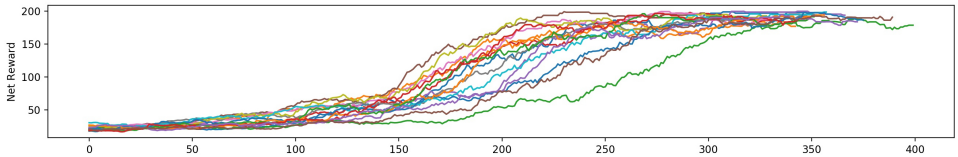
(a) Average performance of 4 agents.



(b) Average performance of 16 agents.



(c) Individual performance of 16 agents for seed $\sigma = 0.1$.



(d) Individual performance of 16 agents for seed $\sigma = 0.0$.

Figure 6.3: Graphs for run of seed based Policy Gradient algorithm (on–policy version) for multiple agents. The top two graphs show average performance of multiple agents, and the bottom two graphs show individual performance for 16 parallel agents. All the graphs are plotted by taking a moving average window of size 25.

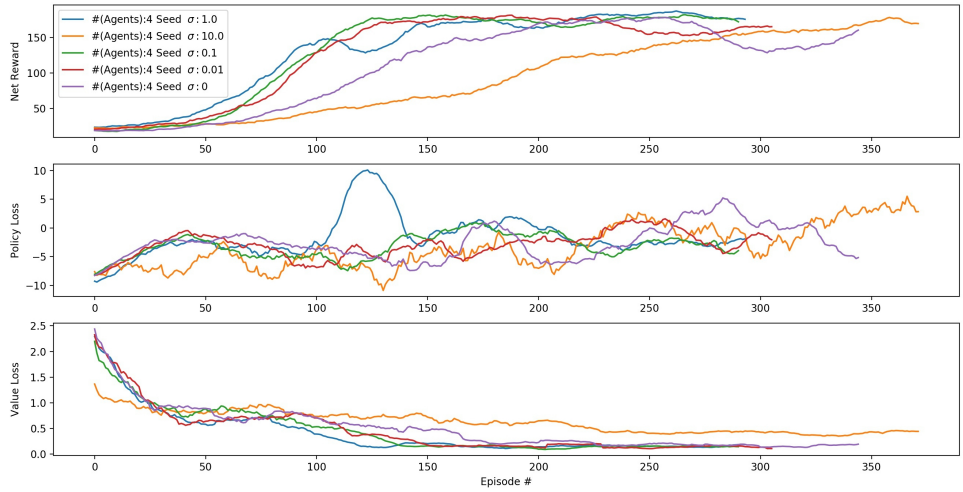
Out of the agents shown in Fig. 6.4a, the group of agents with seed $\sigma = 0.01, 0.1, 1.0$ perform the best, followed by $\sigma = 0$. This shows that increasing the number of agents alone was not sufficient to achieve a faster learning rate. The agent ($\sigma = 0$) is able to get 200 net reward at episode 250, as compared to episode 130 for the better performing agents using seed sampling. This highlights the importance of seed sampling in enabling the agents to explore diversely, leading to varied experiences gathered, and hence faster learning of the environment.

The observation made about seed sampling with 4 agents, however, does not seem to carry to 16 agents. Fig. 6.4b shows that all the agents, except one, exhibit similar performance. We explain this by arguing that the initialization parameters for the function approximators also add randomness (and thereby diversity) to the experience collected by different agents. With 4 agents, this randomness was not sufficient enough to generate diversity in experiences of the different agents, and seed sampling agents were thus more effective. With 16 agents, this intial randomness in parameters is sufficient.

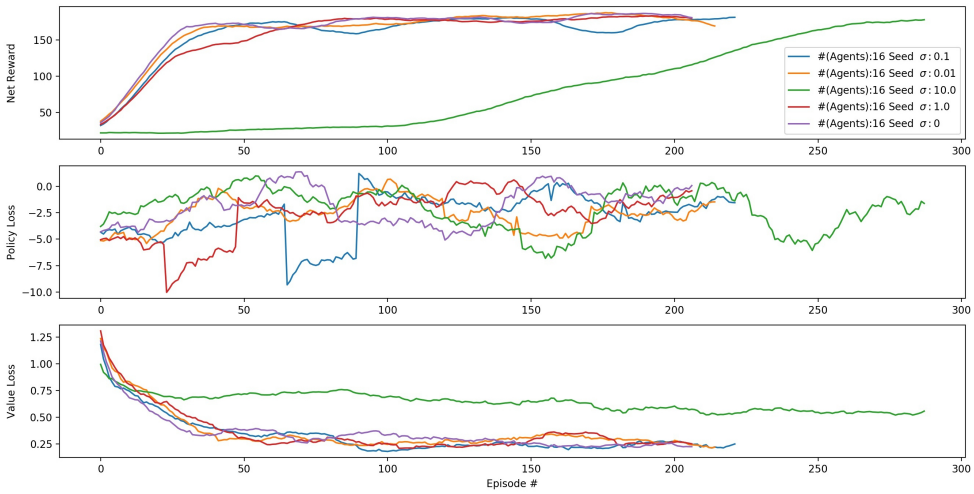
We run another experiment with 16 agents, where we fix the initial paramaters of each of the function approximators to 0.1 (Fig. 6.4c). This form of initialization is considered detrimental for learning as it reduces the assymetry of the neural network. From this experiment we see that the best performance is obtained the agents with $\sigma = 0.01, 0.1$, followed by agent without seed sampling. Also, the agent without seed sampling in this case is unstable as shown by the rewards plot and the value loss plot. On the other hand, all the agents with seed sampling perform stably. Agents with $\sigma = 0.01, 0.1$ also outperform the agent without seed sampling, illustrating the importance of seeds in inducing diversity in agent trajectories.

6.3.1 Off–PAC vs. PPO for off–policy learning

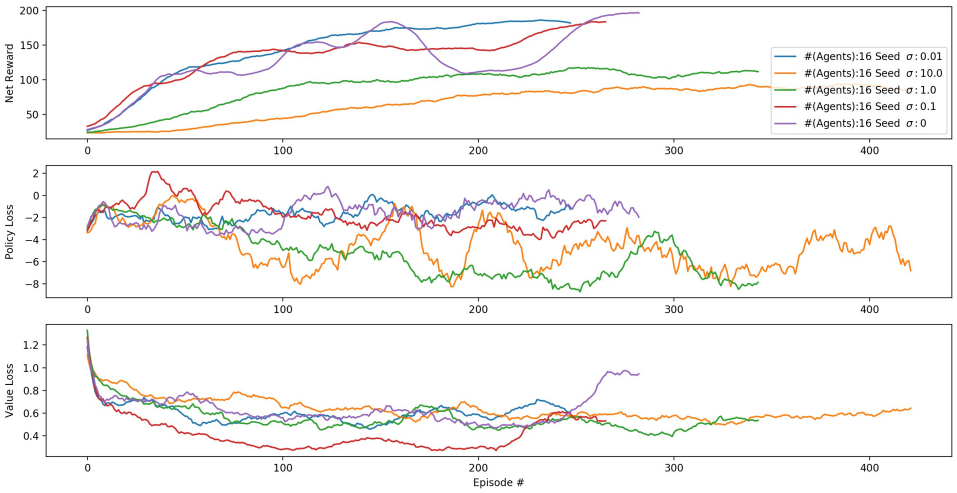
We also ran similar experiments (not shown here) with Off–PAC algorithm (Section 3.3.1) instead PPO. However, Off–PAC was not able to train because of very large IS ratios. When Off–PAC encounters a prohibitively large IS ratio, it performs an update which makes it converge very fast. This early convergence further increases the magnitude of the subsequent IS ratios, and as a result the algorithm fails to learn in the off–policy setting here. PPO effectively handled such high (or low) magnitude IS ratios by clipping them. Further, if tuned carefully, clipping doesn't decrease the number of samples for training drastically: PPO (with $\epsilon = 0.2$) for 8 parallel agents was able to use about 90% of the samples for training, as discussed in Section 5.3.



(a) Average performance of 4 agents, with random agent parameter initializations.



(b) Average performance of 16 agents, with random agent parameter initializations.



(c) Average performance of 16 agents, with fixed agent parameter initializations.

Figure 6.4: Experimental performance of Seed–PG (off–policy Policy Gradient) algorithm on CartPole environment. All the graphs are plotted with a moving average of 25. In the last subfigure, all the agent parameters are initialized with 0.01.

7 Conclusions

In this work we gave a brief introduction to Concurrent Reinforcement Learning. We then explained the method of seed–sampling and its three properties of adaptivity, commitment, and diversity which a concurrent RL agent needs to satisfy in order to explore the environment efficiently.

The method of seed–sampling was, prior to this work, applied in the model based case to tabular MDPs and in the model–free case to value function learning methods. In this work, we proposed two types of learning algorithms: (1) Seed–PG: A model–free method for policy gradient algorithm; and (2) Seed–VIME: A model–based method for the continuous state space MDPs. Seed–VIME is also an extension of the curiosity based VIME (Variational Information Maximization Exploration) algorithm to the concurrent RL case with multiple agents. It is possible to incorporate with Seed–VIME both value based and policy based methods.

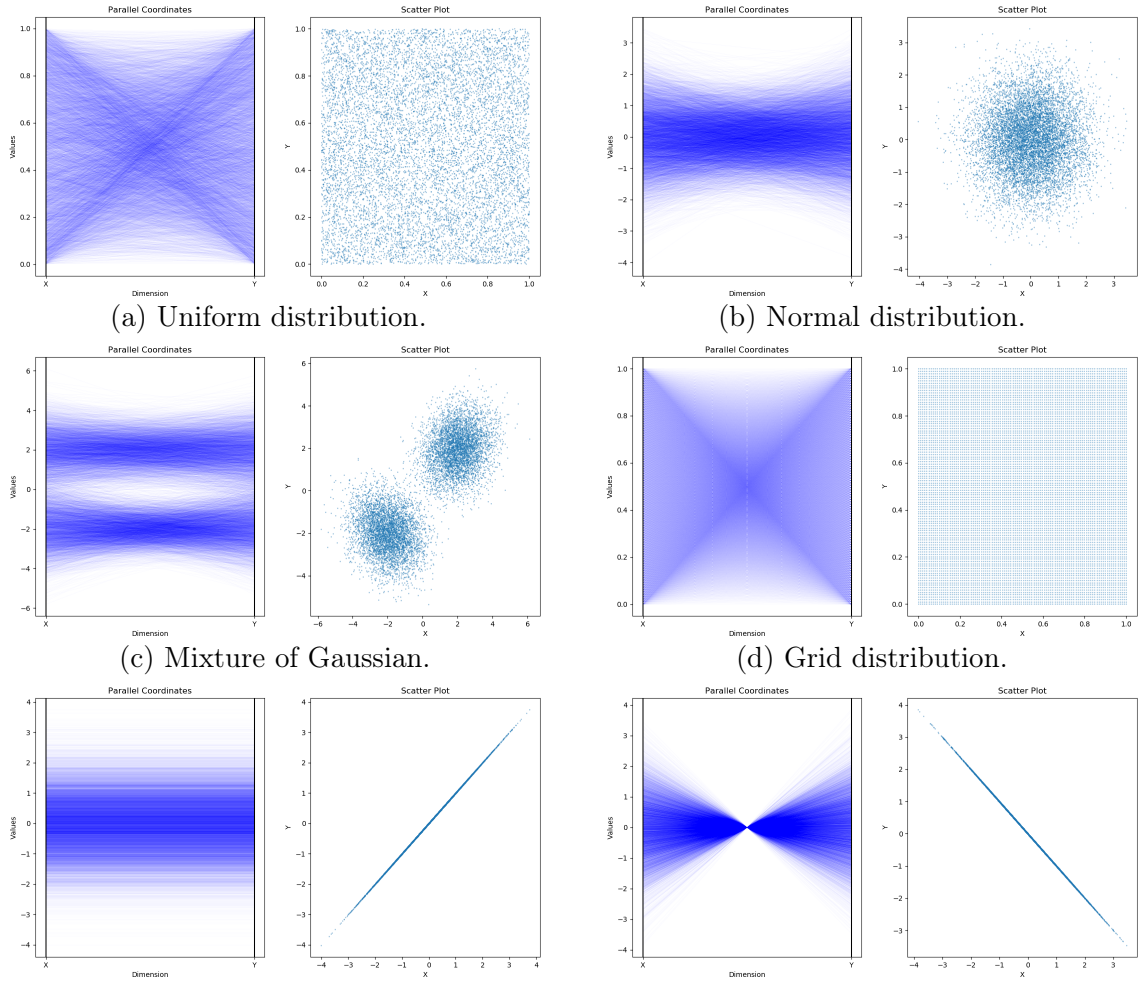
We then experimented with two variants of the seed–PG algorithm: on–policy and off–policy policy gradient. From our experiments we conclude that:

- The introduction of seeds increased state exploration in the case of a single agent.
- On–policy variant of the Seed–PG algorithm is not able to take advantage of the parallel agents. Thus, sharing experience buffer while training both policy and value functions is important, rather than just while training the value function.
- Off–policy variant of seed–PG is able to exploit the parallel agents and its rate of learning increases with the increase in number of parallel agents.
- Seed sampling is able to introduce diversity in the experiences of different parallel agents. Its effect is most pronounced when other sources of diversity (such as randomness introduced from the intial parameterization of policy and value functions, stochasticity in the environment, explorative behavior of the policy) are not available.
- In continuation with the above point, seed sampling can be most useful for increasing diversity when used with a smaller number of agents.

Further work in this direction could involve formulating a theoretical justification into why seed–sampling types algorithms work and how they enable efficient exploration amongst the different agents, and the extension of seed–sampling into a more systematic framework of divide–and–conquer RL, which can explicitly divide the state–action space amongst agents to explore.

A Parallel Coordinates Plot

We visualize the trajectories of agents in our experiments. These trajectories are collection of high dimensional vectors. Visualizing these trajectories sheds qualitative information regarding how much of the state space the agent has explored. This information is useful when we talk about diversity in seed sampling based methods. In order to effectively visualize high–dimensional data on 2–D plots, we make use of parallel coordinates [31]. Parallel Coordinates represents multi–dimensional data by multi–lines. Each dimension of a single data point is plotted on different parallel axes. Then a multi–line connects these parallel axes and this represents a single data point. Fig. A.1 shows various parallel coordinates along with their respective scatter plots for different 2–D distributions.



(a) Uniform distribution.

(b) Normal distribution.

(c) Mixture of Gaussian.

(d) Grid distribution.

(e) Degenerate Gaussian.

(f) Degenerate Gaussian.

Figure A.1: Parallel Coordinate and Scatter Plots for various 2–D distributions. The plots on the left in a sub–figure are the parallel coordinate plots and on the right are their respective scatter plots.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction*. Adaptive computation and machine learning, MIT Press, 1998.

[2] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *Machine Learning: ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, pp. 280–291, 2005.

[3] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural networks," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 1613–1622, JMLR.org, 2015.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[5] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *CoRR*, vol. abs/1611.01224, 2016.

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[7] C. J. C. H. Watkins and P. Dayan, "Technical note q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[9] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pp. 1057–1063, 1999.

[10] T. Degris, M. White, and R. S. Sutton, "Off-policy actor-critic," *CoRR*, vol. abs/1205.4839, 2012.

[11] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[12] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[13] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 1928–1937, 2016.

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[15] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pp. 1889–1897, 2015.

[16] S. Han and Y. Sung, "Dimension-wise importance sampling weight clipping for sample-efficient reinforcement learning," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pp. 2586–2595, 2019.

[17] M. Dimakopoulou and B. V. Roy, "Coordinated exploration in concurrent reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 1270–1278, 2018.

[18] D. Silver, L. Newnham, D. Barker, S. Weller, and J. McFall, "Concurrent reinforcement learning from customer interactions," in *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pp. 924–932, 2013.

[19] M. Dimakopoulou, I. Osband, and B. V. Roy, "Scalable coordinated exploration in concurrent reinforcement learning," *CoRR*, vol. abs/1805.08948, 2018.

[20] H. P. van Hasselt, A. Guez, M. Hessel, V. Mnih, and D. Silver, "Learning values across many orders of magnitude," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 4287–4295, 2016.

[21] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel, "VIME: variational information maximizing exploration," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 1109–1117, 2016.

[22] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933.

[23] D. Russo, B. V. Roy, A. Kazerouni, I. Osband, and Z. Wen, "A tutorial on thompson sampling," *Foundations and Trends in Machine Learning*, vol. 11, no. 1, pp. 1–96, 2018.

[24] G. Hinton and D. van Camp, "Keeping neural networks simple by minimising the description length of weights. 1993," in *Proceedings of COLT-93*, pp. 5–13.

[25] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017.

[26] A. Graves, "Practical variational inference for neural networks," in *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pp. 2348–2356, 2011.

[27] D. Hafner, J. Davidson, and V. Vanhoucke, "Tensorflow agents: Efficient batched reinforcement learning in tensorflow," *arXiv preprint arXiv:1709.02878*, 2017.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[29] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[31] J. Heer, M. Bostock, and V. Ogievetsky, "A tour through the visualization zoo," *Commun. ACM*, vol. 53, pp. 59–67, June 2010.