HYPERPARAMETER TUNING FOR REINFORCEMENT LEARNING WITH BANDITS AND OFF-POLICY SAMPLING

by

KRISTEN HAUSER

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Department of Computer and Data Sciences

CASE WESTERN RESERVE UNIVERSITY

May, 2021

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the dissertation of

Kristen Hauser

candidate for the degree of **Master of Science***.

Committee Chair

Dr. Soumya Ray

Committee Member

Dr. Michael Lewicki

Committee Member

Dr. Harold Connamacher

Committee Member

Dr. Vipin Chaudhary

Date of Defense

February 1, 2021

*We also certify that written approval has been obtained

for any proprietary material contained therein.

Contents

List of Figures

List of Tables

List of Acronyms

A2C Advantage Actor-Critic.

BO Bayesian Optimization.

DDPG Deep Deterministic Policy Gradient.

DQN Deep Q Network.

DR Doubly Robust.

GAE Generalized Advantage Estimator.

GP Gaussian Process.

HOOF Hyperparameter Optimization On the Fly.

HPP Hyperparameter Policy Pair.

HT-BOPS Hyperparameter Tuning with Bandits and Off-Policy Sampling.

IS Importance Sampling.

IW Importance Weight.

KL Kullback-Leibler.

LHS Latin Hypercube Sampling.

MAB Multi-Armed Bandit.

MAGIC Model and Guided Importance Sampling Combining.

MDP Markov Decision Process.

OPE Off-Policy Evaluation.

PB2 Population Based Bandits.

PBT Population Based Training.

PPO Proximal Policy Optimization.

RL Reinforcement Learning.

SAC Soft Actor-Critic.

SIR-PF Sampling Importance Resampling-Particle Filter.

SWUCB Sliding-Window Upper Confidence Bound.

TRPO Trust Region Policy Optimization.

UCB Upper Confidence Bound.

WDR Weighted Doubly Robust.

WIS Weighted Importance Sampling.

Hyperparameter Tuning for Reinforcement Learning with Bandits and Off-Policy Sampling

Abstract

by

KRISTEN HAUSER

The choice of hyperparameters is a critical step in any machine learning experiment. In the RL domain, hyperparameters need to be tuned in an online setting while remaining efficient in terms of the number of samples. While a RL algorithm's performance is sensitive to the configuration of its hyperparameters, there is no consensus on the treatment of hyperparameters in RL. This thesis first surveys the treatment of hyperparameter tuning in current literature. One key idea I exploit is to convert the problem of hyperparameter tuning in RL to a non-stationary multi-armed bandit problem. This is solved with a windowed upper confidence bound algorithm. The second key idea is to reuse samples collected by one configuration to estimate the value of a different configuration to improve sample efficiency. To do this, I use a particle filter to compute off-policy reward estimates. I perform a detailed comparison between several state of the art tuning strategies across four RL algorithms on three different environments. My results show that this approach often produces the best rewards and competitive convergence rates across multiple algorithms and domains compared to other state-of-the-art baselines. This new procedure performs significantly better when the configurations leading to high rewards are rare.

Chapter 1

Introduction

Reinforcement Learning (RL) is a field within artificial intelligence that learns through trial and error. An important subset of RL algorithms, that this work focuses on, are online RL algorithms. Online RL algorithms receive new sequential data points as the model interacts with the environment, either one at a time or in small batches. Online learning is a setting that spans multiple fields within artificial intelligence where algorithms begin with little or no data and acquire more data in a sequential manner. A commonly used subset of algorithms in the online learning setting is supervised learning algorithms. Supervised learning algorithms focus on learning a mapping from inputs to outputs which requires a labeled data set.

One important element of learning algorithms is the configuration of what are known as *hyperparameters*. These are not parameters of the model, but instead refer to the parameters of the learning algorithm itself. Figure 1.1 visualizes an online supervised learning algorithm being fed data by conveyor belts. The learning algorithm is depicted as a black box which is fed data and outputs a model. In this example, the hyperparameters would be knobs on the box.

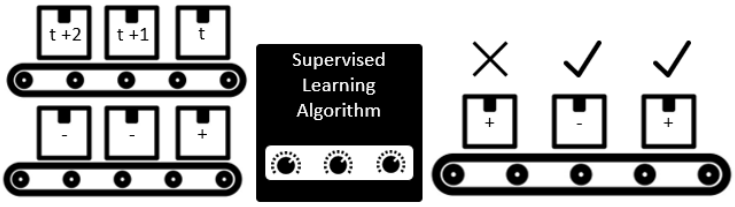

Figure 1.1: Supervised learning where hyperparameters are knobs on the box.

As each data point passes through the box, the black box outputs the model which is used to predict a label. Additionally, the actual label is fed into the box on a separate conveyor belt. This feedback is used to update the model. Turning the knobs on the black box is equivalent to adjusting the hyperparameters. Providing the same data while varying the hyperparameters will produce a different model. In order to produce a model that achieves maximum performance, a configuration for the hyperparameters must be chosen at each iteration that optimizes the performance of the algorithm. This problem is fittingly called *hyperparameter optimization*.

In terms of the black box analogy, the black box is only fed by a single conveyor belt. As the data is fed into the algorithm, the model is updated. The model is then used within the environment to produce more data points which are placed on the conveyor belt. This is illustrated in Figure 1.2.

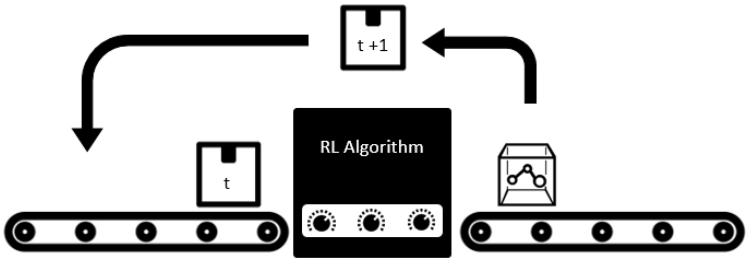

Figure 1.2: A visualization of reinforcement learning as a conveyor belt.

For example, consider the task of a search and rescue as shown in Figure 1.3. The goal of this task is to rescue a person denoted by the target while hazards, such as

fire or water, are marked as X's. A robot can gather experience about this task by navigating the environment and finding the target through interactions. One of these interactions is shown in Figure 1.4. The initial position is provided to the robot, which takes an action. This action is then executed within the maze which produces a new position and reward. The robot then can use that experience to produce a concept that improves the robot's ability to complete the maze at the next attempt.

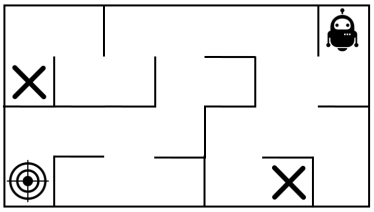

Figure 1.3: A visualization of a search and rescue task.

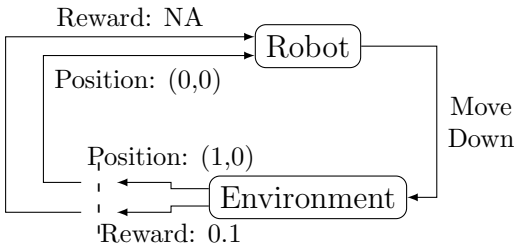

Figure 1.4: Robot abstraction of search and rescue robot-environment interaction.

The model's interaction with the environment creates a significant difference between online supervised learning and RL. The learning process now affects the data set by changing the model which is responsible for generating new data. Hyperparameters dictate important details within a RL algorithm such as how fast the actor, which in the previous example is the robot, should learn or how short-sighted the actor should be. When learning, RL algorithms can show a wide variation in performance under differing hyperparameter configurations.

In order to illustrate this wide variation in performance, an example of Deep Q Network (DQN) on CartPole is used. CartPole is an environment that requires a pole to be balanced on a cart which can be pushed left or right. DQN has three hyperparameters: learning rate, discount rate and exploration rate. A hyperparameter configuration is a set of values, one for each hyperparameter required by the RL algorithm. Ten hyperparameter configurations were generated and run for a fixed number of training iterations. The results of this example are shown in Figure 1.5. The expected reward bounds across hyperparameter configurations are shown as the

shaded area and the mean across configurations is shown as the dotted line. After about a third of the training iterations the range in expected reward is from 7 to 200. This range is actually the reward bounds as defined by the environment. This shows that these 10 hyperparameter configurations show the maximum performance possible for this example. This example emphasizes the importance of hyperparameter tuning in order.

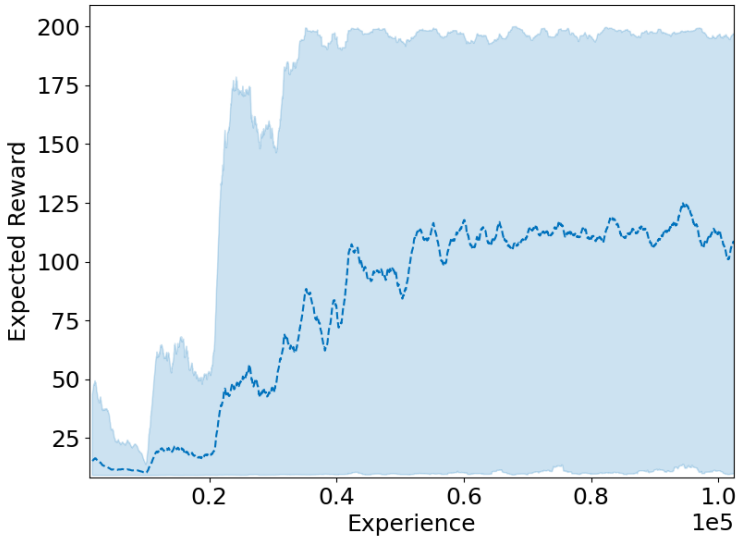

Figure 1.5: A visualization of variance of performance from hyperparameter configurations.

This raises the question: how can we effectively tune hyperparameters for reinforcement learning? The RL setting creates many interesting problems in terms of hyperparameter tuning. The methodologies utilized for optimizing the hyperparameters in other fields are no longer feasible, since the data is time sequenced and not provided to the algorithm as a complete batch at the start of training. The most notable problem with tuning hyperparameters in the RL setting is that the model generates its own data to be used at the next iteration. Since the model is updated using the hyperparameters, the generated data is heavily influenced by the choice of hyperparameters. Thus they need to be carefully chosen at every step in order to

ensure advantageous data is generated at the next iteration. Hyperparameters are pivotal to a RL algorithm's performance and therefore must be tuned.

While this question has been studied for other fields such as supervised learning, there are very few procedures that apply to reinforcement learning. Of the available procedures, most are not sample efficient or they produce a fixed hyperparameter configuration. In this work, I propose a method for tuning hyperparameters in reinforcement learning algorithms. My contributions are:

1. A survey and empirical comparison of existing hyperparameter tuning methodologies for reinforcement learning.

2. A novel algorithm for tuning hyperparameters using multi-armed bandit and off-policy estimations for reinforcement learning algorithms.

3. An evaluation of the presented algorithm for several RL algorithms across multiple environments.

This work demonstrates that at a minimum, the Hyperparameter Tuning with Bandits and Off-Policy Sampling (HT-BOPS) algorithm can be used to tune the hyperparameters of any reinforcement learning algorithm and furthermore provides of guide to when various state of the art tuning methods should be used.

This thesis is organized as follows: In Chapter 2, I present hyperparameter tuning methods in supervised learning, as well as a more formal definition of reinforcement learning. Additionally, I will discuss common RL algorithms and present the necessary background on off-policy estimation and multi-armed bandit problems. In Chapter 3, I motivate the need for tuning in reinforcement learning and describe popular and state of the art approaches for tuning seen in the literature. Chapter 4 formulates the problem of hyperparameter tuning as a multi-armed bandit problem. In Chapter 5, I present results of the described tuning methods across a variety of

algorithms and environments. Finally, in Chapter 6, I summarize my contributions and discuss directions for future work.

Chapter 2

Background and Related Work

This chapter discusses the necessary background and relevant literature for this thesis. I begin with discussing how hyperparameter tuning is treated in supervised learning and some commonly used tuning methods. Then, I formally define reinforcement learning and the general multi-armed bandit problem. Additionally, I will detail several reinforcement learning algorithms used in the experiments that follow. Finally, I will describe techniques for off-policy estimation.

2.1 Hyperparameter Tuning in Supervised Learning

One of the most studied fields in machine learning is supervised learning. As previously mentioned, an important element of learning algorithms is the configuration of hyperparameters. Thus, I begin by examining how hyperparameters are treated within the supervised learning task Additionally, I discuss commonly used tuning algorithms for supervised learning algorithms.

2.1.1 Hyperparameters in Literature

A learning algorithm defines a set of hyperparameters. For an online linear regression problem, a single hyperparameter is used: learning rate, α. The learning rate controls the magnitude of change to the model parameters. A hyperparameter configuration is a set of values that corresponding to each hyperparameter defined by the learning algorithm. So for the online linear regression problem, a hyperparameter configuration would consists of a single parameter but for more complex learning algorithms a set could consists of 5 or more values.

There are two common settings of supervised learning: batch and online. The batch setting trains a model based on all the available data, while the online setting trains a model incrementally by feeding data to the algorithm in a sequential manner. The batch setting is well studied and procedures for hyperparameter tuning are well defined. However, the tuning procedures for the online setting are still being developed.

Batch Setting Hyperparameter tuning in a batch setting is traditionally performed as a loop around cross validation. Cross validation is a method of evaluation and comparing learning algorithms by dividing the dataset in order ensure the model is robust. Since the batch setting has all the data prior to training, the cross validation procedure using the entire dataset is completed for each hyperparameter configuration. Then the hyperparameter configuration which generates the best performance metric is chosen.

Online Setting While the batch setting is well studied, there is insufficient agreement on how to handle hyperparameter tuning in an online setting. A variety of methods have been used, such as manually picking hyperparameter values and fixing them for all datasets. These choices may be the product of a sensitivity analysis,

random trials or an unknown methodology. A common approach is to use tuning approaches in a batch setting first, and then use the hyperparameters found for additional experiments run in the online setting.

2.1.2 Grid and Random Search

Two of the most popular hyperparameter tuning methods are grid search and random search [4]. These methods can be run in parallel or sequential settings and are commonly used in the batch setting of supervised learning tasks. These methods require feedback from a performance metric that is often measured based on cross validation. A commonly used performance metric for supervised learning is accuracy. Both of these tuning methods require a set of models trained using a variety of hyperparameter configurations. Then, the models are evaluated, and the hyperparameter configuration is selected greedily based on the performance metric. These methods differ on how the initial set of hyperparameters is generated.

Grid search requires a finite set of values to be defined for each hyperparameter. The set of hyperparameter configurations is generated by enumerating through each hyperparameter's value set. For example, if a learning algorithm has 2 hyperparameters and each hyperparameter has a finite set of 3 values, then grid search would generate 9 hyperparameter configurations. Grid search is a more exhaustive search mechanism where every combination of hyperparameters is included in the set. The grid search algorithm is described in Algorithm 1.

Algorithm 1 Grid Search

Input: Search space for each hyperparameter, Φ, training data, D
Output: Model f, Hyperparameter Configuration ϕ

1: Generate candidate hyperparameters, ϕ, from enumeration of Φ
2: Split data into training, D_t, and evaluation, D_e, sets
3: **for** $\phi_n \in \phi$ **do**
4: Initialize model, f_n
5: Fit model f_n using ϕ_n and D_t
6: Calculate performance metric of f_n using D_e
7: **end for**
8: Select model, f_m, and hyperparameter configuration, ϕ_m with maximum performance metric
9: **return** f_m, ϕ_m

The algorithm above mentions two key components of machine learning methods: initializing a model, line 4, and fitting a model, line 5. A model is an artifact of an algorithm that maps an input to an output, which can vary in complexity from a linear approximation to a neural network. In terms of a linear model, $f(x) = \vec{m} \cdot \vec{x} + b$, the initialization of the model refers to the initial setting of the parameters, $\vec{m}$ and b. The second component mentioned is fitting a model, which modifies some or all of the parameters within the model to the provided data. Returning to the linear model example, a value would be added to each of the parameters. This value can be calculated via gradient descent which requires a loss function. A common loss function is mean squared error, $L = \frac{1}{n} \sum_{i=0}^{n} (y_i - f(x_i))^2$. Gradient descent requires the first derivative to be calculated with respect to the parameters of the model shown in Equations 2.1 and 2.2 below.

$$\frac{\partial L}{\partial m} = \frac{-2}{n} \sum_{i=0}^{n} -2\vec{x_i}\big(y_i - (\vec{m} \cdot \vec{x_i} + b)\big) \tag{2.1}$$

$$\frac{\partial L}{\partial b} = \frac{-2}{n} \sum_{i=0}^{n} -2\big(y_i - (\vec{m} \cdot \vec{x_i} + b)\big) \tag{2.2}$$

How the parameters of the model are updated is defined by the update rule as follows:

$$\vec{m} \leftarrow \vec{m} + \alpha \frac{\partial L}{\partial m} \tag{2.3}$$

$$\vec{b} \leftarrow \vec{b} + \alpha \frac{\partial L}{\partial b} \tag{2.4}$$

where α is the learning rate which is a common hyperparameter for methods that use gradient descent or its variations. In the remainder of this work, the update rules are defined in terms of the loss functions without the derivatives explicitly taken. Additional hyperparameters can be used within the loss function.

The initialization and fitting of the model described above takes place within a loop. The linear model defines only a single hyperparameter, learning rate. Thus the set of hyperparameter configurations would be the finite set of learning rate values. This set would be iterated over and would result in a model and performance metric value for each. Then the hyperparameter configuration and model with the best performance metric value would be selected.

While grid search requires a finite set of values to be defined for each hyperparameter, random search requires a distribution to be defined for each hyperparameter. Additionally, the number of configurations to generate is required in order to sample from the distributions provided for each hyperparameter. The random search is described in Algorithm 2. While random search is more efficient than grid search, both methods produce a fixed hyperparameter setting and are sample inefficient since no knowledge of previous evaluation of hyperparameter combinations is used in subsequent evaluations.

Algorithm 2 Random Search

Input: Distribution search space for each hyperparameter, Φ, number of hyperparameter configurations, N, training data, D

Output: Model f, Hyperparameter Configuration ϕ

1: Generate hyperparameters configurations, ϕ, from sampling of Φ
2: Split data into training, D_t, and evaluation, D_e, sets
3: **for** $\phi_n \in \phi$ **do**
4: Initialize model, f_n
5: Fit model f_n using ϕ_n and D_t
6: Calculate performance metric of f_n using D_e
7: **end for**
8: Select model, f_m, and hyperparameter configuration, ϕ_m with maximum performance metric
9: **return** f_m, ϕ_m

2.1.3 Bayesian Optimization

Bayesian Optimization (BO) is an optimization method for noisy black-box functions and is commonly applied to hyperparameter optimization [25]. BO is applied to many fields within machine learning but frames the hyperparameter optimization problem as a sequential model-based optimization problem. BO assumes that the objective function is expensive to evaluate. In terms of hyperparameter optimization, the objective function represents the fitting and evaluation of the model.

In order to reduce the use of the objective function, BO utilizes surrogate function which approximates the objective function. The surrogate function is a probabilistic model trained on the past evaluation results. There exist many different methods of constructing this surrogate function that include but are not limited to: Gaussian Processes, Random Forest Regression or Tree Parzen Estimator [30].

The learned surrogate function is then used to construct a selection function which directs sampling to areas where an improvement over the best observation is likely. For hyperparameter tuning, the selection function is used to select a hyperparameter configuration to evaluate. There are also a variety of choices for the selection function: Maximum Probability of Improvement, Expected Improvement or Upper Confidence

Bound. The details of BO are highly dependent on the choice of surrogate and selection functions. Algorithm 3 generally describes the procedure.

Algorithm 3 Bayesian Optimization

Input: Search space for each hyperparameter, Φ, training data, D, training iterations, T, surrogate function, g, selection function, h, objective function, v
Output: Model f, Fixed Hyperparameter Configuration ϕ

1: Initialize g
2: Generate hyperparameters configurations, ϕ, from sampling of Φ
3: Split data into training, D_t, and evaluation, D_e, sets
4: **for** $i = 1, \ldots, T$ **do**
5: Optimize g over h to choose ϕ_i using D_t
6: Evaluate $f_i, x_i = v(\phi_i)$ using D_e
7: Store x_i in X
8: Fit g using X
9: **end for**
10: **return** f_T, ϕ_T

While a variety of choices of surrogate and selection functions are possible, consider the surrogate choice of a Gaussian Process (GP). A GP is a model that constructs a joint probability distribution over the variables by assuming a multivariate Gaussian distribution. This model is initialized and fit with a similar strategy to that as described in Equations 2.1-4. The posterior of the GP is then used to estimate the performance metric given a hyperparameter configuration. The selection function within this example is Probability of Improvement and is defined as:

$$h(x) = \Phi\Big(\frac{\mu(x) - \mu^+}{\sigma(x)}\Big) \tag{2.5}$$

where Φ is the normal cumulative density function, μ and σ are the mean and standard deviation of the surrogate function for given sample x, and μ^+ is the mean of the surrogate, i.e. best performance metric value, seen thus far. On line 3, candidate hyperparameter configurations are evaluated using Probability of Improvement which in turn utilizes the esimates from the GP. The hyperparameter with the largest Probability of Improvement is selected as ϕ_i. Then line 4 evaluates the hyperparameter

configuration using the expensive, objective function which will be used to improve the model and the future hyperparameter configuration choices.

Similar to random search, BO requires a distribution to be defined for each hyperparameter. Unlike grid search or random search, BO keeps track of previous hyperparameter configuration evaluations and uses them to inform later decisions by constructing and updating the surrogate function.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is based on the fundamental problem solving technique which utilizes trial and error. This technique is formulated as a computational approach that solves a problem with numerical feedback from an environment. Actions within an environment produce a numerical reward, which is then used to identify actions that maximize a numerical reward. The general framing of a reinforcement learning problem is described in Figure 2.1.

This framing consists of the agent which is the decision-maker and learner and the environment which consists of everything outside of the agent. The interaction between the environment and agent is represented by discrete time steps, which is denoted by t. An environment defines an action space $\mathcal{A}$ and state or observation space $\mathcal{S}$. At each time step, the agent receives a representation of the environment, $s_t \in \mathcal{S}$, and selects an action, $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of available actions

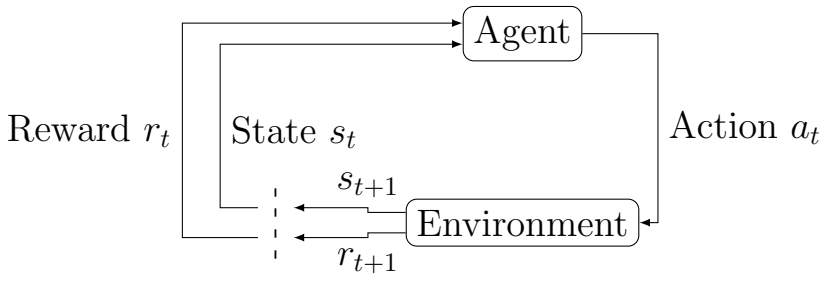

Figure 2.1: Agent-Environment Interaction in Reinforcement Learning

given that state. At the next time step, the agent receives the numerical reward r_t as a result of taking that action. A step is commonly referred to as the tuple that consists of the state, action and reward at a given time. A trajectory consists of a sequence of steps until a terminal condition is reached. A terminal condition can be a fixed length, or horizon, or a condition within the environment which terminates the agent's current path through the environment.

2.2.1 Returns

In the system defined above, an agent's goal is to maximize the reward and a sequence of rewards is defined for a number of time steps. However, the agent generally seeks to maximize the expected return, commonly denoted as G_t at time t. The return can be thought of as the sum of future rewards. Returns can be more easily defined when an environment naturally breaks the sequence of state-action-reward pairs into subsequences, or episodes. However, some tasks continue without limit. This introduces the concept of discounting. In general, discounted returns are used for episodic or continuous tasks and is defined as:

$$G_t = \sum_{k=0}^{N} \gamma_k R_{t+k+1} \tag{2.6}$$

where γ is the discount factor. The discount factor is bounded by $0 \leq \gamma \leq 1$. When $0 \leq \gamma < 1$ and the reward is bounded, this formulation results in a finite value for an infinite sum. The configuration of γ directly influences how short or long sighted the agent is when trying to maximize the rewards.

2.2.2 Policies

One important concept within a RL framework is the policy which defines the agent's way of behaving by mapping states to actions. A policy is comprised of two main com-

ponents: a function approximator and a policy sampler. The function approximator maps the state space to the inputs of the policy sampler. A function approximator must be chosen based on the environment; for example, a tabular function approximator cannot be used for continuous state spaces since it requires an enumeration of all discrete state space descriptors. Additionally, the function approximator approximates different values for different algorithms. For value based algorithms, the function approximator's output is the estimated values of each action at a given state while policy based algorithms output parameters for the policy sampler.The policy sampler can be seen as an exploration strategy.A common policy sampler for value iteration algorithm is the ϵ-greedy strategy. This strategy chooses the maximum reward estimate with $1 - \epsilon$ probability and uniformly samples from the actions with ϵ probability. While a common policy sampler strategy for policy based algorithms is the softmax function. The softmax function normalizes the exponential of each action value to compute a probability of each action.

2.2.3 Markov Decision Process

A RL problem that satisfies the Markov property is called a Markov Decision Process (MDP). The Markov property is as follows:

$$P(R_{t+1} = r, S_{t+1} = s' | S_t, A_t) = P(R_{t+1} = r, S_{t+1} = s' | S_0, A_0, ..., S_t, A_t) \tag{2.7}$$

This property can be summarized as: The future is independent of the past given the present. MDPs are the mathematical basis of an RL problem [27].

Almost all RL algorithms involve estimating value functions in one way or another. Value functions, $V_\pi(s)$, are functions that estimate the expected return of a given state for an agent. The rewards seen by an agent are dependent on the actions by that agent in a given state. The value function assumes the agent follows policy, π, at any

future time steps. Additionally, an action-value function, $Q_\pi(s, a)$, assumes the agent will follow policy, π, after taking action a in state s. For finite MDPs, the Bellman equation for the value and action-value functions are as follows:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s, a)[R(s, a, s') + \gamma V^\pi(s')] \tag{2.8}$$

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} P(s'|s, a)\big[R(s, a, s') + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s', a')\big] \tag{2.9}$$

where π is the policy, a and s are a state-action pair, s' is the next state, P is the state transition probability function, and R is the reward function. The Bellman functions express a relationship between the value of a state or state-action pair and the values of its successor states.

To solve an MDP, an optimal policy, π^*, must be identified. By definition if a policy is optimal, then it also has the optimal value and action-value functions, $V^{\pi^*}(s)$ and $Q^{\pi^*}(s, a)$ respectively. The optimal state and state-value functions are defined as follows:

$$V^{\pi^*}(s) = \max_\pi V^\pi(s) \tag{2.10}$$

$$Q^{\pi^*}(s, a) = \max_\pi Q^\pi(s, a) \tag{2.11}$$

There is no general closed form solution for identifying an optimal policy, although, many iterative approaches have been proposed. Some state of the art iterative approaches are described in Section 2.3.

2.2.4 Multi-Armed Bandit

The Multi-Armed Bandit (MAB) problem is a classic RL problem, whose name comes from a reference to a slot machine. Each bandit is assumed to have n arms, each of which has a stochastic reward distribution. The object is to pull the arms one-by-one

in sequence such that the total reward collected is maximized. The MAB problem can be seen as a simplified MDP, since it does not have state and the actions are defined as pulling a given arm. MAB problems are a classic demonstration of the exploration vs exploitation dilemma. Exploration is intentionally taking sub-optimal actions with respect to the current value function estimate in order to search the solution space for a more optimal solution. Exploration provides information which is valuable, but too much exploration can result in a significant amount of sub-optimal rewards. Since the MAB problem does not have state, many solutions focus on estimating the action-value function defined as $Q(a)$.

The stationary bandit problem assumes that the distribution associated with the reward of each arm does not change over time. The stationary MAB problem is a largely studied problem, with many solutions such as Upper Confidence Bound (UCB)1, ϵ-Greedy and Thompson sampling.

UCB1 The UCB1 algorithm is based on the optimism in the face of uncertainty (OFU) principle by favoring exploration of actions with a high potential to have a better choice [2]. The UCB1 algorithm measures the potential of a choice by computing the upper confidence of the reward value such that the true value of the choice is less than or equal to the potential value with high probability. In UCB variants, the greediest choice is always chosen as such:

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \hat{Q}_t(a) + \hat{U}_t(a) \tag{2.12}$$

where $\hat{Q}_t(a)$ is the estimated value of arm a and $\hat{U}_t(a)$ is the confidence interval of arm a. For UCB1 the estimate arm value is the average of the samples generated from that given arm. The UCB1 algorithm computes the confidence interval using Hoeffding's Inequality. The Hoeffding's inequality is defined for a bounded random

variable X on $[c_1, c_2]$ where $-\infty < c_1 \leq c_2 < \infty$:

$$\mathbb{P}[\mathbb{E}[X] > \bar{X}_t + \hat{U}_t(a)] \leq e^{-\frac{2N_t(a)\hat{U}_t(a)^2}{(c_2-c_1)^2}} \tag{2.13}$$

where $N_t(a)$ is the number of times arm a has been chosen at time t. Solving for the confidence bound results in:

$$\hat{U}_t(a) = \sqrt{\frac{(c_2 - c_1)^2 \log(p)}{-2N_t(a)}} \tag{2.14}$$

where p should be a small probability. For UCB1, the authors chose $p = t^{-4}$. This results in the final formulation of UCB1 as:

$$\hat{U}_t(a) = \sqrt{\frac{2(c_2 - c_1)^2 \log(t)}{N_t(a)}} \tag{2.15}$$

An important note on the result associated with UCB1 is that it has theoretical guarantees in terms of regret. Regret is defined as the difference between the reward that is possible to achieve, and the reward that was actually achieved [24]. UCB1 has logarithmic regret over n trials while not requiring any preliminary knowledge. The UCB1 algorithm is a popular solution to the multi-armed bandit problem due to this strong theoretical guarantee. It is important to note that the regret bound includes an infinite sum and for the regret bound to be convergent, p, must be chosen such that $\lim_{p\to\infty} = 0$. This requirement is satisfied due to the author's choice of $p = t^{-4}$.

For example, consider a three armed bandit problem where each of the three arms have an associated reward distribution as shown in Figure 2.2. When an arm is pulled a sample is generated from this distribution. The UCB1 example calculation is shown under the reward distributions after several arms have been pulled. Each arm must be pulled at least once to so that confidence interval is defined. The confidence interval of an arm at a time is denoted as $\hat{U}_t(a)$ in the previous equations.

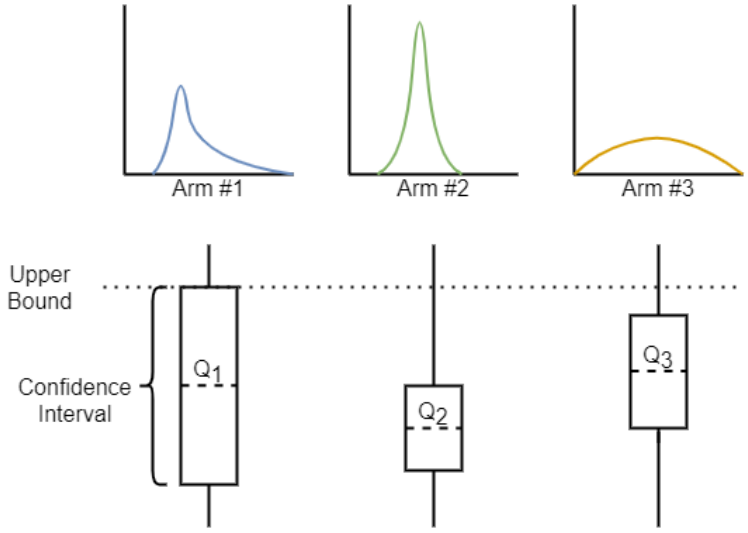

Figure 2.2: Upper Confidence Bound for a 3-armed Bandit

The expected values of each arm, $\hat{Q}_t(a)$ is computed by averaging the history of samples from that arm which is represented as the Q values and the dashed line in Figure 2.2. The confidence interval is shown as the box around the expected value of each arm. The arm that defines the upper bound is the selected arm which in this case is Arm 1 which is then pulled to generate a sample. For the next iteration the arm value and confidence interval will need to be recalculated for the first arm since the samples for that arm have changed. This is likely to reduce the confidence interval for Arm 1 since another sample was generated. The confidence interval decreases as the number arm samples increases because the confidence in the accurarcy of the estimate increases.

ϵ-Greedy The ϵ-Greedy algorithm is a simple algorithm that is based on random exploration. The action-value function is estimate by simply averaging the reward seen from each arm as follows:

$$\hat{Q}(a) = \frac{\sum_i^t r_i I(a_i = a)}{N_t(a)} \tag{2.16}$$

where t is the total number of trials, a_i and r_i are the action and reward of the i^{th} trial, $I(a_i = a)$ is an indicator function for whether the specific arm was chosen and $N_t(a)$ is the use count of the specific arm. At each selection, this algorithm draws p from a uniform distribution, then chooses a random action if $p < \epsilon$, otherwise greedily selects the action based on the action value estimate. This algorithm is not used within this thesis due to having no theoretical guarantees in regards to regret.

Thompson Sampling Thompson Sampling is based on the "benefit of the doubt concept". Thompson Sampling chooses an arm based on the likelihood that the arm is optimal [1]. Thompson Sampling is a Bayesian approach which models the posterior distribution, $P(a|H_t)$. Each iteration samples the posterior for each arm and selects the arm with the largest posterior probability. This algorithm requires prior distributions to be defined and thus an appropriate likelihood and conjugate prior pair must be chosen. This algorithm is not used within this thesis due to its reliance on pre-defined distributions. The form of these distributions would likely be unknown in real world applications.

2.3 Reinforcement Learning Algorithms

In this section, I discuss several algorithms, value and policy based, used for reinforcement learning. Reinforcement learning algorithms can be categorized by several attributes. RL algorithms can first be described as model-free or model-based. Model-based algorithms learn a model of the environment which is then used to take actions within the environment. A model is a function that estimates a received output [27]. For example, some model based approaches estimate the reward function of the environment. Model-free algorithms do not construct a model of the environment or any of its components. Model-free algorithms can be further divided into value-based and policy-gradient variants. Value-based algorithms learn a value function in order

to derive a policy on which to act on while policy-gradient algorithms directly learn a policy [27]. Additionally, algorithms can be described as either on-policy or off-policy. Two variations of policies can exist within an algorithm: an update policy and a behavior policy. The update policy is the policy used by the agent when learning the optimal policy. The behavior policy is used by the agent when interacting within the environment. On-policy algorithms use the same policy for both the update and behavior policies, while off-policy algorithms use different policies for each [27]. Since off-policy algorithms can use samples when the update and behavior policies are different, this allows for samples to be trained on multiple times and thus they are seen as more sample efficient than on-policy methods.

2.3.1 Q-Learning

Q-Learning is a value-based algorithm based on off-policy temporal difference learning. Temporal difference learning is an approach to learning how to predict a value that depends on unknown, future values. Temporal difference learning gets its name from utilizing the differences, or changes, in predictions over time. Value-based algorithms focus on learning the value function or action-value function. Q-Learning focuses on learning the action-value function which is a direct approximation of the optimal action-value function through temporal difference learning. A policy is then derived using the action-value approximation. Q-Learning is considered off-policy since it is learning the value of a policy that is different from the one being followed [27]. The Q-Learning update rule is as follows:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \Big(\frac{1}{2} \big(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a) \big)^2 \Big) \tag{2.17}$$

where s is the state, a is the action, r is the reward, s' is the next state, γ is the discount factor, α is the learning rate and Q_θ is the function approximator parame-

terized by θ. Q-Learning is restricted to discrete action spaces because they are based on the discrete-space version of Bellman's equation, However, Q-Learning can handle discrete or continuous state spaces based on the choice of function approximator. Within this thesis, Deep Q Network (DQN) [14] is used to represent the Q-Learning algorithm. DQNs use deep neural networks as their function approximators as opposed to a tabular method or classic polynomial functions.

DQN has three hyperparameters that are tuned within this work: learning rate, α, discount factor, γ and ϵ or ϵ_d. Learning rate determines the step size at each training iteration which dictates how fast or slow the model parameters are adjusted. Discount factor dictates how far ahead in time an algorithm looks and is further described within Section 2.2.1. The last hyperparameter tuned is ϵ, initial exploration rate, or ϵ_d, exploration decay. The ϵ-greedy policy sampler is used within this implementation of DQN, which can utilize a fixed ϵ or a schedule. Both policy sampler styles are used within the tuning. When a schedule is used the initial ϵ is set to 1 and a ϵ_d is sampled, while when a fixed ϵ is used the ϵ is sampled and ϵ_d is set to 1.

2.3.2 Advantage Actor-Critic

Advantage Actor-Critic (A2C) is an on-policy, actor critic method [13]. As previously mentioned, methods are typically categorized into value iteration or policy iteration while actor critic methods generate estimates for value and policy functions. Value based methods derive a policy from the value estimate, while policy gradient methods estimate the policy directly. For policy gradient methods, a value function may still be used to learn the policy, but it is not required for action selection [27]. Policy gradient methods aim to solve the "simpler" problem since it removes a step by computing the policy directly and is more easily expandable to continuous action spaces. For continuous action spaces, consider the use of an ϵ-greedy sampling strategy, the value estimate can be computed but an additional optimization step is required to find the

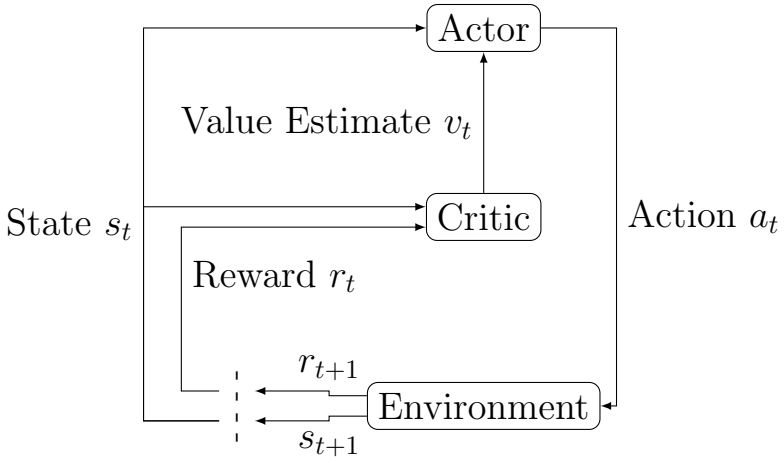

Figure 2.3: Actor Critic-Environment Interaction in Reinforcement Learning

optimal action. This optimization would have to occur for every action, which is not tractable in some cases. Policy gradient methods avoid this problem entirely by learning the policy directly and thus is commonly seen as a "simpler" problem.

Within actor-critic methods, to estimate both of these functions, the model is split in two: the critic which computes a value approximation and an actor which computes a policy as described in Figure 2.3. Policy gradient methods commonly can suffer from higher variance and so a critic was introduced to facilitate more stable learning by improving the value estimates.

In the case of A2C, the critic learns the value function while the policy gradient update then utilizes the advantage. The advantage is defined as the difference between the state-action value and the state value, this reduces the variance of the estimate [26]. The definition of advantage is shown in Equation 2.18. The state-action value can be decomposed into the current reward and the state value for the next state which is shown in Equation 2.19. Within A2C, the advantage is calculated using the return and the state value approximator, critic. This formulation is more commonly

know as a return as discussed in Section 2.2.1 and this form is shown in Equation 2.20.

$$A_t = Q(s_t, a_t) - V(s_t) \tag{2.18}$$

$$= r_t + \gamma V(s'_t) - V(s_t) \tag{2.19}$$

$$= G_t - V(s_t) \tag{2.20}$$

where s is the state, a is the action, r is the reward, s' is the next state, $V(s)$ is the value function and $Q(s, a)$ is the action value, and R is the return for the given step. In terms of RL, the advantage represents how much better an action is compared to all other actions at a given state, while the value function represents how valuable a given state is for accomplishing the given task. As previously mentioned, adding a critic improves the stability of learning, while utilizing the advantage function also reduces the variance of the estimator. The A2C update rules are as follows:

$$\theta_V \leftarrow \theta_V + \alpha \nabla_{\theta_V} \Big(\frac{c_v}{2} \big(G_i - V_\theta(s_i) \big)^2 \Big) \tag{2.21}$$

$$\theta_\pi \leftarrow \theta_\pi + \alpha \nabla_{\theta_\pi} \Big(- \log \big(\pi_\theta(a_i | s_i) \big) \big(G_i - V_\theta(s_i) \big) - c_e \texttt{entropy}(\pi_\theta) \Big) \tag{2.22}$$

where s is the state, G is the discounted return, α is the learning rate, c_v is the value loss coefficient, c_e is the entropy loss coefficient and V_θ and π_θ are the critic and actor function approximators, respectively, parameterized by θ. The actor update rule is of a similar form to the Q-Learning rule, while the policy update rule utilizes the log probability of the action given the state, advantage and an additional entropy term. On-policy models are found to lead to highly-peaked policies for a given state which means an action or small region of actions have significantly higher probability than the rest of the action space. This occurs because it is easy for both the actor and critic to over optimize a small region of the policy corresponding to limited portion of the environment. To reduce this problem, the entropy term is added to the loss to

encourage exploration [13].

A2C has five hyperparameters that are tuned within this work: learning rate, α, discount factor, γ, entropy coefficient, c_e, value coefficient, c_v, and Generalized Advantage Estimator (GAE) trade-off factor, λ. The first two hyperparameters are common across all RL algorithms used within this work and are described in Section 2.3.1. The entropy and value coefficients are used to scale their affect on the update rule and are usually set to < 1. The entropy coefficient dictates how much exploration is desired. The value coefficient is required when model parameters are shared between the policy and value models. The A2C implementation used within this works shares model parameters between the policy and value models. Thus, the value coefficient dictates the affect of the value model loss on the policy model. The final hyperparameter for A2C is λ dictates the trade-off between bias and variance for GAE [22]. When $\lambda = 1$, the GAE is equivalent to classic advantage as described in Equation 2.18. GAE utilizes the temporal difference residual which is defined in 2.19 which is referred to as δ_t at time t. Thus GAE is defined as:

$$A_t^{GAE} = \sum_{i=o}^{\infty} (\gamma\lambda)^i \delta_{t+i} \tag{2.23}$$

where γ is the discount factor, λ is the GAE trade-off factor, and δ is the temporal difference residual. This formulation is commonly used in order to reduce variance or bias within the estimates which can improve sample efficiency of policy gradient methods. Since OpenBaselines uses this formulation, the classic advantage estimate is replaced with the GAE and thus λ was added to the hyperparameter list.

2.3.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy, actor-critic algorithm based on A2C [23]. PPO defines the critic update in the same manner as A2C while a modifica-

tion is made to the actor update. Policy gradient method performance can be greatly impacted by the learning rate. Too small of a learning rate and the task may not converge within reasonable time constraints or may converge to a sub-optimal solution, while too large of a learning rate may cause large model updates resulting in oscillating performance. PPO focuses on stability within the learning process while allowing for multiple iterative updates by restricting the gradient for each update. While two versions of PPO were proposed in the original paper, PPO-Clip was implemented and used throughout this work. The other variant of PPO utilizes Kullback-Leibler (KL) divergence, which measures how different two probability distributions are. The KL divergence constraint is used to restrict the iterative updates, this methodology is similar to that found in Trust Region Policy Optimization (TRPO) [21].

PPO-Clip was chosen to be used in this work since it outperformed the KL divergence variant. PPO-Clip variant focuses on stabilizing learning by clipping the weighted advantage at each update. At each iteration in PPO-Clip, the weighted advantages are clipped via the following rule:

$$P_{clip}(\pi_{\theta,k}) = \min\left(\frac{\pi_{\theta,0}(a_i|s_i)}{\pi_{\theta,k}(a_i|s_i)}\big(G_i - V_\theta(s_i)\big), \texttt{clip}\Big(\frac{\pi_{\theta,0}(a_i|s_i)}{\pi_{\theta,k}(a_i|s_i)}\big(G_i - V_\theta(s_i)\big), -\epsilon_a, \epsilon_a\Big)\right) \tag{2.24}$$

where s is the state, g is the discounted return, π_0 is the policy that generated the samples, and V_θ and π_θ are the critic and actor function approximators, respectively, parameterized by θ. The action probability ratio between the current policy and the policy used to generate the samples are used to weight the advantage. This concept is based off of Importance Sampling (IS) which will be further discussed in Section 2.4.1. An additional variant of PPO-Clip also clips the critic term so the difference between the initial value estimate and the new value estimate is within ϵ_c. Different clipping values can be used for the actor and critic and thus can result in an extra hyperparameter.

PPO-Clip completes k updates of policy within a learning iteration using the following update rule:

$$\theta_V \leftarrow \theta_V + \alpha \nabla_{\theta_V} \Big(\frac{c_v}{2} \big(G_i - V_\theta(s_i) \big)^2 \Big) \tag{2.25}$$

$$\theta_\pi \leftarrow \theta_\pi + \alpha \nabla_{\theta_\pi} \Big(- P_{clip}(\pi_{\theta,k}) - c_e \texttt{entropy}(\pi_{\theta,k}) \Big) \tag{2.26}$$

where s is the state, G is the discounted return, α is the learning rate, c_v is the value loss coefficient, c_e is the entropy loss coefficient,π_0 is the initial policy, and V_θ and π_θ are the critic and actor function approximators, respectively, parameterized by θ. Since PPO is based on A2C the entropy term is used again to promote exploration.

PPO has seven hyperparameters that are tuned within this work: learning rate, α, discount factor, γ, entropy coefficient, c_e, value coefficient, c_v, GAE trade-off factor, λ, actor clipping fraction, ϵ_a, and critic clipping fraction, ϵ_c. The first two hyperparameters are common across all RL algorithms used within this work and are described in Section 2.3.1, while the next three hyperparameter are from A2C and described in detail in Section 2.3.2. The two unique hyperparameters for PPO are the clipping ratios, one for the actor and another for the critic. The actor clipping ratio value is used in Equation 2.24.

2.3.4 Soft Actor-Critic

Soft Actor-Critic (SAC) is an off-policy, actor-critic algorithm [7]. While A2C and PPO encourage exploration by adding the entropy term to the policy update, SAC takes an approach based on entropy-regularized RL. Entropy-regularized RL encourages exploration while remaining successful by not only maximizing rewards but also maximizing the entropy of the policy. Encouraging exploration of the state space improves the data collected, allows the policy to capture multiple forms of good policies and prevents premature convergence to a local optimum [7].

The version of SAC used in this thesis is restricted to continuous action spaces and uses a fixed entropy coefficient. Previous methods have focused on learning a single value function and the policy, while SAC learns three functions: policy, π_θ, a value function, V_θ, and a soft action-value function, Q_θ. The update rules are as follows:

$$\theta_V \leftarrow \theta_V + \alpha \nabla_{\theta_V} \Big(\frac{1}{2} \big(V_\theta(s_i) - Q_\theta(s_i, a_i) + c_e \texttt{entropy}(\pi_\theta) \big)^2 \Big) \tag{2.27}$$

$$\theta_Q \leftarrow \theta_Q + \alpha \nabla_{\theta_Q} \Big(\frac{1}{2} \big(Q_\theta(s_i, a_i) - r - \gamma V_\theta(s_{i+1}) \big)^2 \Big) \tag{2.28}$$

$$\theta_\pi \leftarrow \theta_\pi + \alpha \nabla_{\theta_\pi} \Big(\log \big(\pi_\theta(a_i|s_i) \big) - Q_\theta(s_i, a_i) \Big) \tag{2.29}$$

where s is the state, a is the action, r is the reward, α is the learning rate, c_e is the entropy loss coefficient, Q_θ, V_θ and π_θ are the three function approximators parameterized by θ. It is important to note that in the policy update rule, since the action space is continuous and assumed to have a Gaussian distribution, the gradient of the Gaussian distribution must also be included in the update.

High entropy of a policy encourages exploration by assigning equal probabilities to actions that have nearly equal Q values. This framework also ensures that a particular action is not selected repeatedly to exploit an inconsistency in the Q value function approximation. This feature overcomes a common brittleness problem seen in many other algorithms.

SAC has three hyperparameters that are tuned within this work: learning rate, α, discount factor, γ, and entropy coefficient, c_e,. The first two hyperparameters are common across all RL algorithms used within this work and are described in Section 2.3.1, while the next hyperparameter, c_e, is described in detail in Section 2.3.2.

2.4 Off-Policy Evaluation

Off-Policy Evaluation (OPE) aims to estimate the value of a policy based on data collected by a different policy or set of policies. This is important topic within RL, since running a new policy may be expensive or risky. There are two general approaches to OPE: importance sampling based and model based. This section will take a look at some commonly used and state of the art OPEs.

2.4.1 Importance Sampling

One family of OPE methods is based on the popular, model free, un-biased estimator, Importance Sampling (IS) [8]. IS has numerous variations and among the most popular is Weighted Importance Sampling (WIS) [19]. The foundation of this family of estimators is the Importance Weight (IW) is defined as:

$$IW(\pi_b, \pi_e, \tau) = \frac{\prod_{i=0}^{T} \pi_e(a_i|s_i)}{\prod_{i=0}^{T} \pi_b(a_i|s_i)} \tag{2.30}$$

where a_i and s_i are the actions and states for a trajectory, τ, of length T, and π_b and π_e are the behavioral and evaluation policies, respectively. The behavior policy is the policy that was being followed when the trajectory was generated, while the evaluation policy is the policy being evaluated in comparison. The main idea behind IS is to weight the rewards generated by the behavioral policy based on the relationship of how likely the reward is under the evaluation policy. The IS and WIS estimate the value of a policy using a history of trajectories, H, is as follows:

$$IS(\pi_b, \pi_e, H) = \frac{\sum_{\tau}^{H} IW(\pi_b, \pi_e, \tau) R_\tau}{len(H)} \tag{2.31}$$

$$WIS(\pi_b, \pi_e, H) = \frac{\sum_{\tau}^{H} IW(\pi_b, \pi_e, \tau) R_\tau}{\sum_{\tau}^{H} IW(\pi_b, \pi_e, \tau)} \tag{2.32}$$

where R_τ, is the cumulative reward of trajectory τ.

This is a commonly used estimator due to its simplicity and un-biased nature. Additionally, the estimate of the value is independent of the state space size. However, this estimator suffers from high variance in long horizon domains or as the evaluation policy, π_e, and behavior policy, π_b diverge. Furthermore, IS and its variants are typically used to estimate a stationary distribution based on a history of samples from another stationary distribution.

2.4.2 Blended Estimators

While IS based approaches can suffer from high variance, model-based approaches can suffer from high bias. Two state of the art solutions that blend importance sampling and model based approaches are: Doubly Robust (DR) and Model and Guided Importance Sampling Combining (MAGIC). These methods are not purely importance sampling methods nor model based methods, and thus are commonly referred to as blended estimators.

The first approach is DR, which generates a model of the MDP to reduce variance of the un-biased estimate of the value produced by IS [10]. This estimator is considered doubly robust, since its estimates are unbiased if the model is accurate or the behavioral policy is known. The DR estimator makes use of a variant of IS known as step wise IS. The DR estimate can be defined recursively while the value function and action-value functions are estimated via model fitting, such as regression using a separate data set.

MAGIC is a new blended estimator which uses an extension of the DR estimator that utilizes WIS instead of IS. This new extension is referred to as Weighted Doubly Robust (WDR) [28]. MAGIC uses WDR as the off-policy return estimate for each step. DR requires a separate data set from the evaluation data set to train the model, while MAGIC also requires a large data set to train.

Both of these state of the art approaches assume that a single policy generates

data, which can be used to estimate additional policies. The work within this thesis focuses on model free approaches and thus these methods are not used.

Chapter 3

Hyperparameter Tuning In Reinforcement Learning

This chapter will present a review of the state of the art hyperparameter tuning methods in reinforcement learning literature. Generally, hyperparameter tuning methods can be categorized as sequential searches, parallel searches and gradient based methods. Sequential searches use the results of the previous configuration to choose the next, while parallel searches reduce run time but require more resources. Gradient based methods frame hyperparameters as trainable and are then updated via back propagation through a meta-objective function.

In this chapter, I will discuss some popular, state of the art tuning methods as well as describe how tuning method were adopted from the supervised learning task. This topic is very popular in current literature and numerous papers have been published within the past few months. The popularity of published papers exemplifies the interest within the community of this topic and its importance.

RL is different from other fields of artificial intelligence since the choice of hyperparameters affects the data generated at the next time step. As previously shown, this can lead to high variation in performance. High variation can be caused by

adjusting a single hyperparameter. For example, an un-tuned hyperparameter can cause an excessive or an inadequate amount of exploration. Either of these cases can cause a significant time delay in solving the task at hand. Delayed or sub-optimal performance can be extremely costly in time and resources, and thus it is important to not only tune hyperparameters but also tune them in an efficient manner.

3.1 General Hyperparameter Tuning Methodology

Hyperparameter tuning methods for RL generally have the same fundamental inputs, outputs and main structure. The inputs, outputs and high level procedure of a tuning method for RL is described in Algorithm 4. The first input of a tuning method is a RL algorithm which defines the hyperparameters that will be tuned using this procedure. Then, the search space needs to be defined for each hyperparameter defined by the RL algorithm. Additionally the number of hyperparameter configurations to generate and the number of training iterations to complete need to be specified. There are two components to the output of a tuning method, the first of which is a maximum policy. This is a policy that produced the maximum reward during the tuning procedure. The hyperparameter schedule that was used to generate this policy is the second component to the output of a tuning method for RL.

Algorithm 4 General Hyperparameter Tuning for Reinforcement Tuning

Input: RL learning algorithm, L, search space, Φ, training iterations, T, number of hyperparameter configuration, N, minimum training iterations
Output: Policy π, Hyperparameter Schedule H

$\phi \leftarrow$ Generate N hyperparameters configurations from sampling of Φ
$P, L \leftarrow$ Initialize policy/policies and learner
for $i = 0, \ldots, T$ **do**
    $P \leftarrow \texttt{update}(\phi, P, L)$
    $V, \pi, H \leftarrow \texttt{evaluate}(\phi, P, H)$
    $P, \phi \leftarrow \texttt{exploit}(\phi, P, V)$
end for
return π, H

The first step of the general procedure generates the hyperparameter configurations using the search space and the number of hyperparameter configurations which occurs on line 1. The initialization of policy/policies and the learner is found on line 2. The policy may be a set or a singleton depending on the tuning method.

Then the tuning iterative procedure has three main steps: Update, Evaluation, and Exploitation. The first step is the update step which gathers data and updates the policies associated with each hyperparameter configuration or generates new policies and is shown on line 4. Each tuning method defines a single policy or set of policies to operate on, then the update step defines how a policy is generated for each hyperparameter configuration or how the set of policies are updated. How a policy is updated is defined by the RL algorithm and requires three components in order to generate a new updated policy. The three required components are: the initial policy, the hyperparameter configuration, and the training data. A Hyperparameter Policy Pair (HPP) associates a hyperparameter configuration with a policy. This tuple satisfies two of the three requirements for a policy update. The persistence of multiple policies allow for more diverse exploration of the solution space which in turn is likely to avoid local maximum within the solution space. The second step is the evaluation step which evaluates the hyperparameter configurations and selects the best and is shown on line 5. The final step is the exploit step which employs a strategy to take advantages of well performing hyperparameter configurations and is shown on line 6. The implementations of these steps various across tuning method.

In order for fair comparison between tuning methods, any experience used within the tuning methods is recorded, which includes training and evaluation interactions with the environment. Additional evaluation occurs outside of the training loop in order to track progress of each tuning method at various experience thresholds. The experience generated from the additional evaluation does not contribute to the experience required for the tuning method since the tuning method does not have

access to the data generated from this experience.

3.2 Supervised Learning Tuning Adoption

Three hyperparameter tuning methodologies for supervised learning were described in Section 2.1. These tuning methods are typically run in a batch setting where the complete training dataset is required. In order to apply them to RL algorithm, a training budget is defined where the RL algorithm will run for a specified number of training iterations. In order to define these methods for RL we will describe how each method defines the update, evaluation and exploit step.

In general, the performance metric used for these two methods is the average cumulative reward across evaluation trajectories. When evaluating a policy in RL, the average cumulative reward is calculated for several evaluation trajectories. A trajectory is collection of steps until a terminal condition is reached where each step is a tuple that consists of a state, action, and reward. The difference between training and evaluation trajectories is based on how an actions are selected at time steps. If actions are sampled from the probability distribution defined by the policy, the trajectory is known as a training trajectory. However, if the most likely actions are always selected, the trajectory is known as an evaluation trajectory. In other words, a training trajectory was generated using a stochastic policy while an evaluation trajectory is generated under a deterministic policy.

Grid and Random Search The generation of the hyperparameter configurations remains the same as the supervised learning variant. Each of these methods use HPP so a policy is initialized for each hyperparameter configuration. The update step consists of a loop over each HPP. The associated policy to each hyperparameter configuration is used to generate an evaluation trajectory which is then used to update the policy. The evaluation step is only completed during the last training iteration.

Evaluation trajectories are generated and then the average cumulative reward is used to identify the best HPP and thus hyperparameter configuration. The exploit step for these methods is essentially an empty function. The final result is a policy and associated fixed hyperparameter configuration.

Bayesian Optimization The update step for BO for RL begins with generating training data and an updated policy using the objective function and the currently selected hyperparameter configuration. The objective function in this case represents generating training data and updating the policy. The result of the objective function is the policy. This single policy is used across iterations. The evaluation step then evaluates this single policy and the evaluation in stored in a history. Then, the surrogate function is fit using the history generated by the evaluation. The surrogate function is optimized over the selection function in order to choose the next hyperparameter configuration. The exploit function for this method is again an empty function. The final result is a policy and associated hyperparameter schedule that was used to generate that policy.

While these previously discussed methods are able to be applied to RL algorithms, they are sample inefficient due to the large number of samples required from multiple training runs. Thus, hyperparameter tuning for RL algorithms requires more complex, state of the art methods.

3.3 Offline Reinforcement Learning

Offline RL is RL applied to previously recorded data, and has been used when gathering data from a policy may be risky or expensive [16]. It has been shown offline RL is also sensitive to hyperparameter choices as well as the choice of algorithm itself. It is important to note that more classical approaches, like those discussed Section 2.1, can be applied to these problems since all of the training data is known prior.

3.4 Population Based Training

A modern, flexible technique, known as Population Based Training (PBT) [9] has been applied successfully in numerous domains. PBT is a combination of sequential and parallel search methods. PBT is highly flexible, since the exploration and exploitation methods are user-defined. The exploit function is used to improve policy performance while the explore function is used to generate new or update existing hyperparameter configurations. PBT utilizes HPP by associating a hyperparameter configuration with a policy. The update step generates training data using the policy from each HPP then updates the policy using that training data. This step is described in Algorithm 5.

Algorithm 5 PBT Update Step

Input: Hyperparameter Configurations, ϕ, Policy set, P, RL learning algorithm, L
Output: Policy Set P

1: **for** $n = 0, ..., len(P)$ **do**
2: Sample training data set, D, from P_n
3: Update policy P_n using ϕ_n and D
4: **end for**
5: **return** P

PBT's evaluation step consists of running evaluation trajectories then computing the average cumulative reward. The hyperparameter configuration is then chosen by selection the maximum cumulative reward. This step is described in Algorithm 6. It is important to note that evaluation trajectories are generated for each hyperparameter configuration which in turn generates a significant amount of experience every time this step is executed. However the experience generated by the evaluation trajectories is not commonly recorded in literature. Within this work, any experience that is utilized by a tuning method is counted toward the experience regardless of whether or not the experience was used to update the policy.

Algorithm 6 PBT Evaluate Step

Input: Hyperparameter Configurations, ϕ, Policy set, P, Hyperparameter Schedule, H
Output: Value Estimates, V, Selected Policy, π, Hyperparameter Schedule, H

1: **for** $n = 0, ..., len(P)$ **do**
2: Evaluate value of P_n and store in V_n
3: **end for**
4: $M \leftarrow \underset{m}{\text{argmax}}\, V$
5: $\pi \leftarrow P_M$
6: $H \leftarrow H \cup \phi_M$
7: **return** V, π, H

The final step of of the general hyperparameter tuning method is the exploit step and is described in Algorithm 7. The exploit step for PBT consists of the execution of user defined exploit and explore functions. A common exploit method, and the one used throughout this thesis, is to copy the function approximator values held within a policy of the top 20% of the hyperparameter configurations to the bottom 20%. The policy chosen from the top 20% is sampled uniformly from the configurations. This exploit function allows for under performing configurations in a phase of training to not be hindered in the next phase of training. The explore method used within this work is to redraw a set of hyperparameter values to be used. After the exploit and explore steps are completed, the worst performing hyperparameter configuration and associated values are replaced with new or related configurations.

Algorithm 7 PBT Exploit Step

Input: Hyperparameter Configurations, ϕ, Policy set, P, Value Estimates, V
Output: Hyperparameter Configurations, ϕ, Policy set, P

1: $\phi, P = \texttt{exploit}(\phi, P, V)$
2: $\phi, P = \texttt{explore}(\phi, P, V)$
3: **return** ϕ, P

Previously discussed tuning methods, such as grid or random search, produce a fixed hyperparameter configuration to be used during all of training. However, it has been shown that hyperparameters need to dynamically change during learning [5].

Hyperparameters can control how much an actor explores the domain or exploits its knowledge, which can vary depending on the phase or training the actor is in. The wrong parameters in a phase of learning can significantly hinder learning. This tuning methodology is the first that generates a hyperparameter schedule instead of a fixed configuration. All of the remaining tuning methodologies discussed in this thesis will generate a hyperparameter schedule. A summary of PBT's implementation of each step is described in Table 3.1.

Step	PBT
Update	• Generate training data from each policy for all HPPs • The policy in each HPP is updated using the associated training data and hyperparameter configuration
Evaluate	• Generates evaluation trajectories and calculate average cumulative reward • Selects HPP with maximum reward
Exploit	• Execute exploit function is used to improve policy performance • Execute explore function is used to get new or update existing hyperparameter configurations

Table 3.1: PBT Step Definitions

3.5 Population Based Bandits

Population Based Bandits (PB2) is an expansion to the PBT algorithm where the custom exploration function of PBT is framed as a batch Gaussian process bandit optimization problem that utilizes a time-varying function [17]. By framing the exploitation function in this way, this method has strong theoretical guarantees due to the regret bound. PB2 defines the i^{th} hyperparameter configuration as x_t^i at time t. The goal of PB2 is to maximize the final reward function, $F_t(x_t)$, by maximizing a time varying black-box function, $f_t(x_t)$. In order to do this, the cumulative regret,

$r_t(x_t)$, is minimized as follows:

$$maxF_t(x_t) = \max \sum_{t=1}^{T} f_t(x_t) = \min \sum_{t=1}^{T} r_t(x_t) \tag{3.1}$$

This black-box function is approximated by a Gaussian process, $f_t \sim GP(\mu_t, k)$, where μ_t is a mean function and k is a kernel. Thus, $f_t(x'_t)$ follows a Gaussian distribution where the mean and variance are defined as:

$$\mu_t(x') = \mathbf{k}_t(x')^T(\mathbf{K}_t + \sigma^2\mathbf{I})^{-1}\mathbf{y}_t \tag{3.2}$$

$$\sigma_t^2(x') = k(x', x') - \mathbf{k}_t(x')^T(\mathbf{K}_t + \sigma^2\mathbf{I})^{-1}\mathbf{k}_t(x') \tag{3.3}$$

PB2 requires all data generated from each setting to be stored and used to train the Gaussian process model. Then, the next hyperparameter configuration can be chosen using the following rule:

$$x_{t,b} = \underset{x \in D}{\operatorname{argmax}}\, \mu_{t,1}(x) + \sqrt{\beta_t}\sigma_{t,b}(x) \tag{3.4}$$

PB2 uses all of the main steps of the general tuning method for RL as defined in the original PBT. However, the explore function is no longer user defined and can vary. The explore function is replaced with the formulation in Equation 3.4. PB2 utilizes existing data to generate more informed decisions on the selection of the new hyperparameter configurations. These decisions were also shown to have sub-linear regret guarantees. However, this procedure is not as sample efficient since it requires parallel runs of each hyperparameter configuration just as the original formulation of PBT requires.

3.6 Meta-Gradient RL

All of the previously mentioned tuning methods require multiple training runs in order to optimize the hyperparameters. Within the RL domain, training runs can be computationally expensive or even impossible if a simulator does not exist. Meta-Gradient is a significantly more sample efficient method that only requires one training run [29]. Meta-Gradient RL is a gradient-based tuning method for hyperparameters. The hyperparameter updates are calculated by computing the gradient of the update with respect to the hyperparameters [29]. This requires the use of a differentiable meta-objective. As previously noted in Section 2.2, within an RL problem the function approximator is parameterized by θ, which is updated at an interval using the common rule:

$$\theta' = \theta + \nabla L(\tau, \theta, \phi) \tag{3.5}$$

where L is the RL algorithm defined update rule, τ is a sequence of experience consisting of states, actions, and rewards, and ϕ is the hyperparameters configuration for the algorithm.

Meta-Gradient RL then calculates the gradient $\frac{\partial \theta'}{\partial \phi}$ which is used as an indicator of how the hyperparameters affect the function approximator parameters. Additionally, Meta-Gradient RL requires a performance measurement of the update function approximator. To do this, a meta-objective is defined as $J'(\tau', \theta', \phi')$, where τ' is a subsequent and independent sequence of experience. By fixing a hyperparameter configuration, ϕ', when evaluating the resultant form is used:

$$\frac{\partial J'(\tau', \theta', \phi')}{\partial \phi} = \frac{\partial J'(\tau', \theta', \phi')}{\partial \theta} \frac{d\theta'}{d\phi} \tag{3.6}$$

The authors note that the parameters form an additive sequence that results in

the gradient being able to be accumulated in an online fashion as such:

$$\frac{d\theta'}{d\phi} = \frac{d\theta}{d\phi}\left(I + \frac{\partial f(\tau, \theta, \phi)}{\partial \theta}\right) + \frac{\partial f(\tau, \theta, \phi)}{\partial \phi} \tag{3.7}$$

Additionally, the authors note that $\frac{\partial f(\tau,\theta,\phi)}{\partial \theta}$ is cumbersome and difficult to compute and thus is estimated with an accumulative trace z. This results in the following estimate:

$$\frac{d\theta'}{d\phi} = \mu z + \frac{\partial f(\tau, \theta, \phi)}{\partial \phi} \tag{3.8}$$

This formulation introduces a parameter μ. This additional parameter alleviates the requirement for a fixed hyperparameter configuration from Equation 3.7. The $\mu \in [0, 1]$ decays the accumulated trace within Equation 3.8. A choice of 0 results in an algorithm that selects hyperparameters based on only the immediate effect. Thus, the final hyperparameter update rule is then defined as follows:

$$\phi \leftarrow \phi + \beta \nabla \left(\frac{\partial J'(\tau', \theta', \phi')}{\partial \theta'} \frac{d\theta'}{d\phi}\right) \tag{3.9}$$

This tuning method operates on a single policy. However, in terms of the general hyperparameter tuning method for RL, Meta-Gradient's update and evaluate steps are highly coupled. After training data is generated by the single policy, the update of the policy and the choice of the hyperparameter configuration is selected at the same time by calculating the gradient of each using the training data. Additionally, there is an classic evaluation step since only one hyperparameter configuration is used which is produced from the current hyperparameter configuration and the gradient. Finally, there is no exploit step since only a single policy is used and a single candidate hyperparameter configuration.

3.7 Hyperparameter Optimization on The Fly

Hyperparameter Optimization On the Fly (HOOF) is another method that only requires a single training run in order to tune hyperparameters. HOOF is a simple and efficient tuning method that greedily selects the hyperparameter configuration at each iteration based on an off-policy estimate of each candidate policy [18]. The method of generating hyperparameter configurations is flexible, but the original method used was uniform sampling within a continuous search space for the hyperparameter configurations.

In terms of the general hyperparameter tuning method for RL, the update step is described in Algorithm 8. The update step for HOOF generates training data form the single policy since this tuning method operates on a single policy. Then, the initial policy and associated training data is paired with each hyperparameter configuration in order to generate a candidate policy.

Algorithm 8 HOOF Update Step

Input: Hyperparameter Configurations, ϕ, Policy, P, RL learning algorithm, L
Output: Policy Set P

1: $\pi \leftarrow P$
2: Sample training data set, D, from π
3: **for** $n = 0, ..., len(\phi)$ **do**
4: Generate candidate policy P_n using π, ϕ_n and D
5: **end for**
6: **return** P

The evaluate step, which is described in Algorithm 9, utilizes an off-policy evaluation method known as Weighted Importance Sampling (WIS) then the hyperparameter configuration with the maximum estimate reward is selected. HOOF then uses importance sampling to calculate off-policy estimates of the value of each candidate policy. Since IS is known to have high variance, HOOF utilizes the Kullback-Leibler (KL) constraint for first order methods. KL divergence measures how different two probability distributions are. In this case, HOOF does not allow two policies to di-

verge significantly which keeps the estimates informative, since the relative ordering of the policies has much lower variance. Any hyperparameter configurations generate a policy that does not satisfy this constraint are not able to be selected.

Algorithm 9 HOOF Evaluate Step

Input: Hyperparameter Configurations, ϕ, Policy set, P, Hyperparameter Schedule, H
Output: Value Estimates, V, Selected Policy, π, Hyperparameter Schedule, H

for $n = 0, ..., len(P)$ **do**
    Estimate value of P_n using WIS and store in V_n
    Compute $\leftarrow KL(P_n||\pi)$ and store in k_n
end for
if RL algorithm is a first order method **then**
    $M \leftarrow \underset{m}{\text{argmax}}\, V$ s.t. $k < \epsilon$
else
    $M \leftarrow \underset{m}{\text{argmax}}\, V$
end if
$\pi \leftarrow P_M$
$H \leftarrow H \cup \phi_M$
return V, π, H

Finally, the exploit step which is described in Algorithm 10. HOOF's implementation of this step consists of discarding of all other candidate policies that are not selected. This ensures only a single policy persists across iterations.

Algorithm 10 HOOF Exploit Step

Input: Hyperparameter Configurations, ϕ, Policy set, P, Value Estimates, V
Output: Hyperparameter Configurations, ϕ, Policy set, P

$P \leftarrow \pi$
return ϕ, P

A summary of HOOF's implementation of each step is described in Table 3.2. HOOF is sample efficient compared to previous tuning methods because of the reuse of training data to evaluate the candidate policies. Additionally, it is unique since it only has a single policy persist across iterations.

Step	HOOF
Update	• Generate training data from the policy • Generate candidate policies for each hyperparameter configuration
Evaluate	• Off-Policy Evaluation-Weighted Importance Sampling • Selects hyperparameter configuration with maximum estimated reward
Exploit	• Discards all candidate policies not selected

Table 3.2: HOOF Step Definitions

3.8 Empirical Evaluation

Two of the previously described tuning methods are used for comparison in these results, HOOF and PBT. HOOF was chosen because of the single run nature of the algorithm and its focus on application to reinforcement learning algorithms. Additionally, in the original presentation of the algorithm it was shown to out perform the meta-gradient tuning method. While PBT requires multiple runs, it was chosen due to its popularity. While variations of PBT have been proposed such as PB2, the original algorithm was chosen to minimize overhead and prior knowledge. PBT's multiple training iterations were run in a sequential manner to facilitate a more fair comparison.

A random baseline results were also produced. The random baseline was produced by completing a training run for all fixed hyperparameter configurations for each environment and RL algorithm. The median expected reward was calculated after every training iteration, then the fixed hyperparameter configuration that is most similar to the median value was selected as the random baseline. This curve is representative of the expected behavior if a random hyperparameter configuration from the set is chosen.

Domains All of the empirical evaluations completed within this thesis utilize three different domains. All of the environments used are from OpenAI Gym or PyBullet

Gym. PyBullet Gym is an open source implementation of the OpenAI MuJoCo environments.

- Cartpole - This OpenAI Gym environment is comprised of a cart and a pole hinged to the cart [3]. The cart is allowed to move along a friction-less track by applying either left or right force to the cart. Initially pole is upright, and the goal is to prevent it from falling over. An episode is terminated when the angle of the pole exceeds 12°, the cart position is $> \pm 2.4$, or after 200 steps. Each step receives a reward of +1.

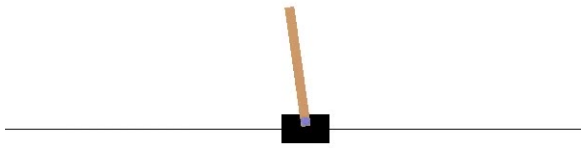

Figure 3.1: Frame of Open AI Gym's CartPole Environment

- LunarLander - This OpenAI Gym environment is comprised of a lander and a landing pad. The lander moves through the environment by do nothing or firing the right, left or main engine. The landing pad is always at (0,0), and fuel is infinite during an episode. An episode is terminated at landing or after 1000 steps. Upon crashing or landing the lander receives an additional reward of -100 or +100 points, respectively. Each leg ground contact is +10 while firing the main engine is -0.3 points each step. The default version of this environment utilizes a shaping function that is added to the reward during training and evaluation. The environment was modified so the shaping function is only used within training.

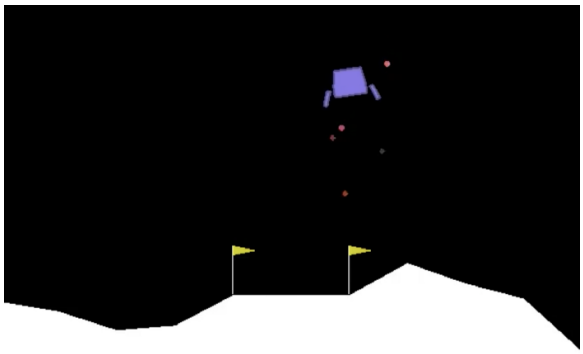

Figure 3.2: Frame of Open AI Gym's LunarLander Environment

- Reacher - This PyBullet Gym environment consists of target ball and a robotic arm that consists of two links and two joints. The joints are actuated by applying torque. Initially the arm is straight to the right and the target is randomly placed within its field of reach. An episode is terminated when the robotic arms tip reaches the target or after 250 steps. The reward at each step is the sum of the negative distance from tip to target and negative norm of the torque applied.

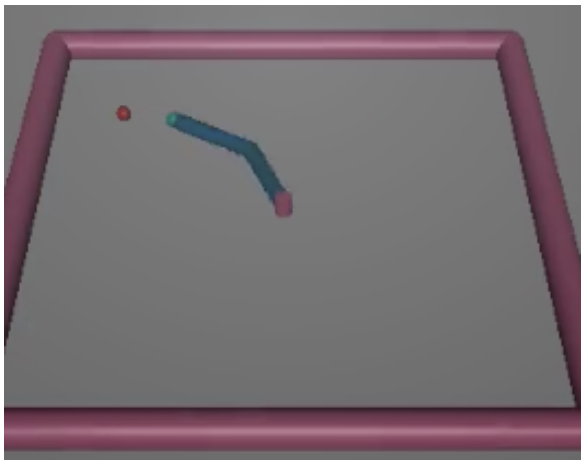

Figure 3.3: Frame of PyBullet Gym's Reacher Environment

Methodology Each tuning algorithm was provided with the same hyperparameter configurations to choose from across tuning methods. A discrete set of values were

defined for each hyperparameter then a hyperparameter configuration was chosen by sampling a value for each hyperparameter required for the given RL algorithm.

Uniform random sampling for each hyperparameter value set is a commonly used technique to generate hyperparameter configurations. However, this can result in similar configurations. For example if a hyperparameter configuration consists of 3 values, two hyperparameter configurations can be generated where 2 of the 3 values are exactly the same if sampling from a finite set. This typically results in one of the hyperparameter configurations being slightly better than the other which results in one of the hyperparameter configurations being essentially meaningless throughout the tuning process. In order to avoid this, a sampling method that maximizes the diversity of the hyperparameter configurations was required.

Thus, the sampling method used to generate the hyperparameter configurations was Latin Hypercube Sampling (LHS). While random sampling generates new points without taking into account previous samples, LHS aims to spread the sampled values more evenly across all dimensions. LHS evenly partitions the sample space for each dimension into region then generates samples random within each N-dimensional space. This ensure the samples resemble the probability distribution of each dimension even with a few number of samples.

For example, consider two discrete parameters each with four values and four samples need to be generated. The samples generated by random sampling are shown in Table 3.3 and the samples generated by LHS are shown in Table 3.4. LHS samples such that each row and column will only be represented once. LHS guarantees variability while random sampling does not. This is emphasized since the third row contains not samples while the second row contains two. Thus, LHS was chosen to generate the most diverse possible set of hyperparameters in order to avoid a significant portion of similarly performing hyperparameter configurations.

Additionally, due to PBT's explore step, which normally requires perturbation or

	X		
X			X
		X	

Table 3.3: Random Sampling

X			
	X		
			X
		X	

Table 3.4: Latin Hypercube Sampling

re-sampling of hyperparameters, PBT operates on half of the hyperparameter configurations. Then during the exploit step, PBT draws from the full hyperparameter set. This allows each algorithm to choose from the full set of hyperparameters such that no single tuning method receives an advantage by accessing different hyperparameter configurations the other methods.

Metrics After each hyperparameter configuration selection, ten evaluation trajectories are run and the median average reward is reported. Additionally, each trial was completed five times with varying random seeds. The maximum cumulative reward is recorded for each trial, and the median across trials is plotted vs experience. Experience includes any interaction from the environment related to learning or the tuning process.

Hyperparameters Four RL algorithms were used throughout this thesis: Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), Soft Actor-Critic (SAC), and Deep Q Network (DQN). The hyperparameters tuned for each RL algorithm is described in Table 3.5. Additionally, descriptions for each of the hyperparameters for an algorithm can be found in the referenced section.

RL Algorithm	Tuned Hyperparameters	Description Section
DQN	γ, α, ϵ or ϵ_d	Section 2.3.1
A2C	γ, α, c_e, c_v, λ	Section 2.3.2
PPO	γ, α, c_e, c_v, λ, ϵ_a, ϵ_c	Section 2.3.3
SAC	γ, α, c_e	Section 2.3.4

Table 3.5: Hyperparameters Tuned for Each RL Algorithm

To ensure a fair comparison, all tuning strategies utilize the same ranges of hyperparameter values. The set of hyperparameters provided to each tuning method varies by trial but are drawn using LHS a the beginning of each trial. Ranges for all parameters are shown below in Table 3.6.

Hyperparameter	Search Space	Set Count
γ	0.1, 0.2, ..., 0.9, 0.95, 0.99	11
α	1e-6, 2.5e-6, 5e-6, ..., 5e-4, 7.5e-4, 1e-3	13
c_e	0, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2	6
c_v	0.25, 0.3, 0.4, ..., 0.7, 0.75	7
λ	0.1, 0.2, ..., 0.9, 0.95, 0.99	11
ϵ_a	0.1, 0.15, ..., 0.35, 0.4	7
ϵ_c	0.1, 0.15, ..., 0.35, 0.4	7
ϵ	0.05, 0.1, 0.15, 0.2	4
ϵ_d	0.9, 0.95, 0.99, 0.999	4

Table 3.6: Search Space for Hyperparameters Used for All Experiments

It is important to note that not all hyperparameters for a RL algorithm were tuned. There exists some hyperparameters, such as batch/buffer size and model structure that were not tuned. All RL algorithm utilized full connected, dense, neural network models but the number of layers and number of nodes per layer were not tuned.

3.8.1 Environment Categorization

A good way to characterize a domain is to calculate the fraction of hyperparameter configurations from the random baseline set that lead to 90% of the optimal reward within N iterations. Since the hyperparameter configurations were generated to ensure diversity, a high fraction of high performing configurations would generally imply that the environment can be solved with a more greedy method. A lower fraction would imply that the environment may require a delicate balance between exploration and exploitation. A summary of the fractions of hyperparameter configurations that are within 90% of optimal is shown in Table 3.7. The random baseline was not able

to be completed for DQN on LunarLander due to time constraints.

Learning Algorithm	Environment	Within 90% Optimal
CartPole	A2C	94.00%
	PPO	94.00%
	DQN	100.00%
LunarLander	A2C	1.33%
	PPO	2.67%
Reacher	A2C	7.00%
	PPO	7.00%
	SAC	1.00%

Table 3.7: Fraction of high performing hyperparameter configurations

CartPole has a high fraction of high performing hyperparameter configurations while the LunarLander and Reacher environments can be seen as significantly more sensitive to hyperparameter configuration choices since their fraction is considerably lower. A high fraction of high performing hyperparameter configurations generally mean that a more greedy method such as HOOF would perform well, while lower fraction environments may require another solution.

3.8.2 Hypotheses

In this section, I present results to test the following hypothesis:

1. HOOF will outperform PBT and the random baseline on average for policy gradient methods.

2. PBT will outperform the RANDOM baseline for value-based methods.

A tuning strategy can outperform another in two ways: asymptotic reward and convergence rate. The asymptotic reward is defined as the policy with largest average cumulative reward found by a tuning strategy which would presumably be the maximum reward of the environment if enough training experience is gathered. The convergence rate is how fast the tuning strategy is able to learn the policy that has

the maximum reward of the environment. Within this work, the convergence rate is inferred from the experience required to reach various thresholds of the maximum reward of the environment. Tuning strategies that require less experience to reach a threshold have a faster convergence rate.

Hypothesis One: Performance on Policy Gradient Methods Three policy gradient methods are used throughout this work: PPO, A2C and SAC. SAC is used for continuous environments while the other two are used for both discrete and continuous environments.

The A2C results are visualized in Figure 3.4. These results show that HOOF outperforms PBT and the random baseline for each environment in terms of convergence rate and have equivalent or better performance with respect to asymptotic reward. The most noticeable performance improvement is on LunarLander, where both PBT and the random baseline barely show improvement while HOOF has already solved the environment.

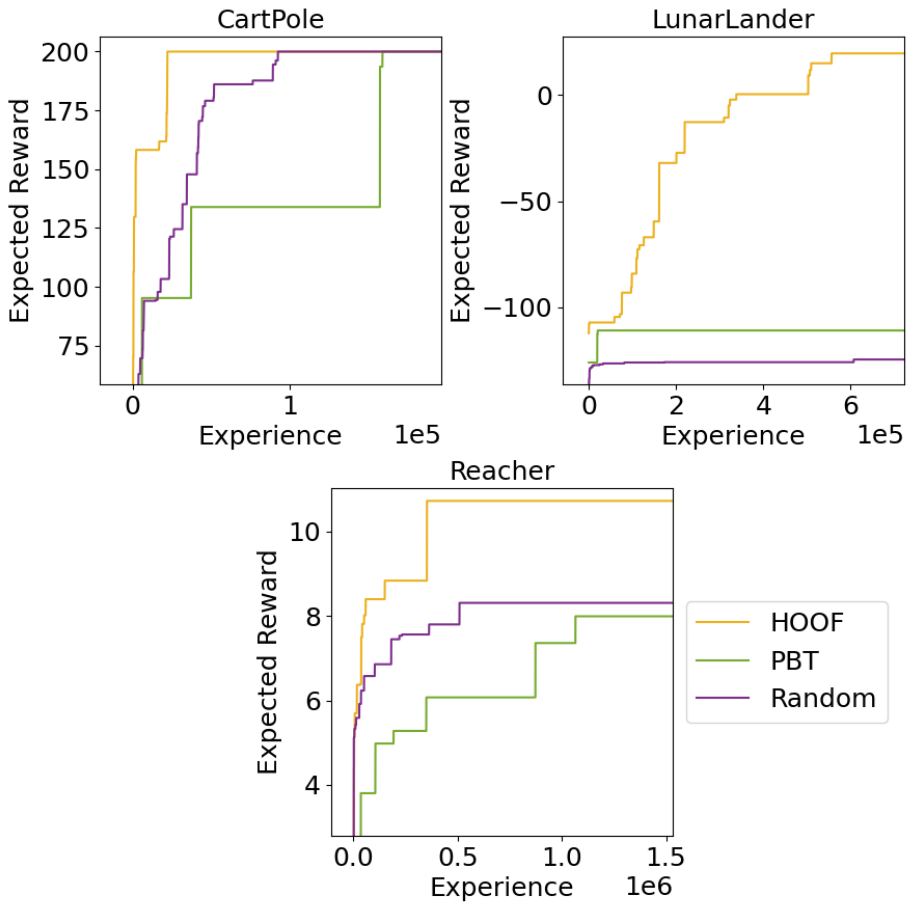

Figure 3.4: Performance of HOOF and PBT compared to random baseline for A2C

The result for PPO are shown in Figure 3.5. Again, HOOF outperform PBT and the random baseline for each environment. While the random baseline is better for PPO than A2C on LunarLander, PBT continues to show marginal improvement while HOOF have already solved the environment.

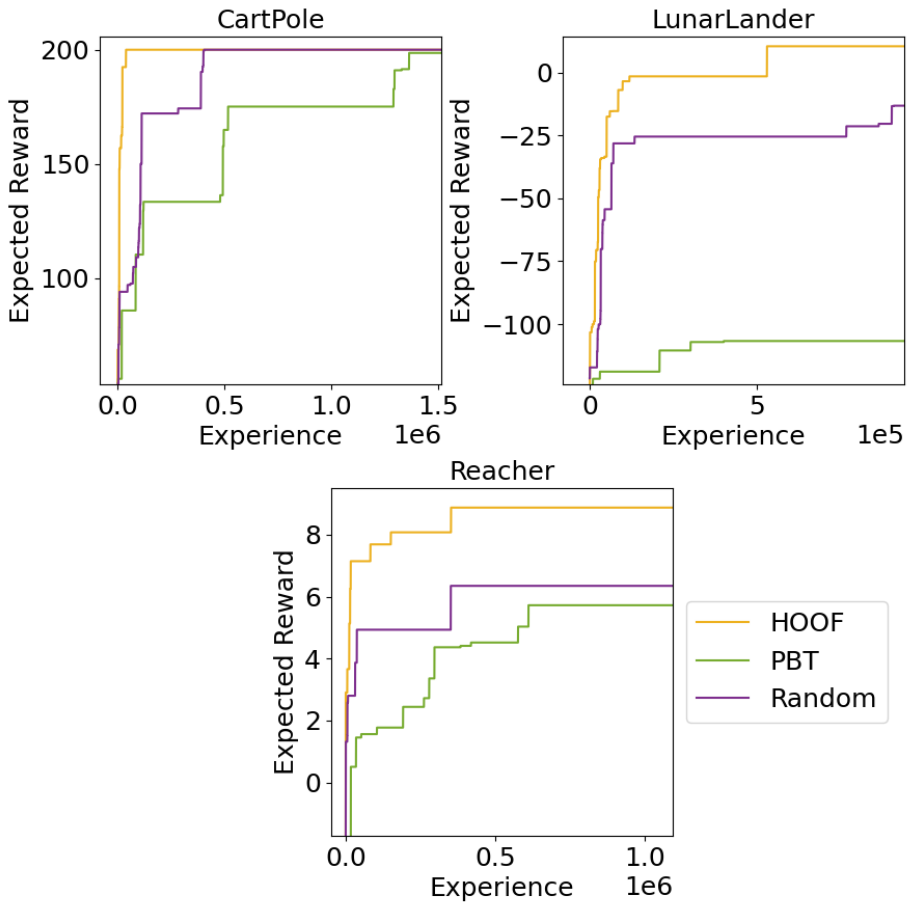

Figure 3.5: Performance of HOOF and PBT compared to random baseline for PPO

The final policy gradient method is SAC which was run on the continuous environment, Reacher. The SAC results are shown in Figure 3.6. Again, HOOF outperforms PBT and the random baseline. However, for this RL algorithm, PBT is more comparable to the random baseline, and outperforms the random baseline during the early stages of learning.

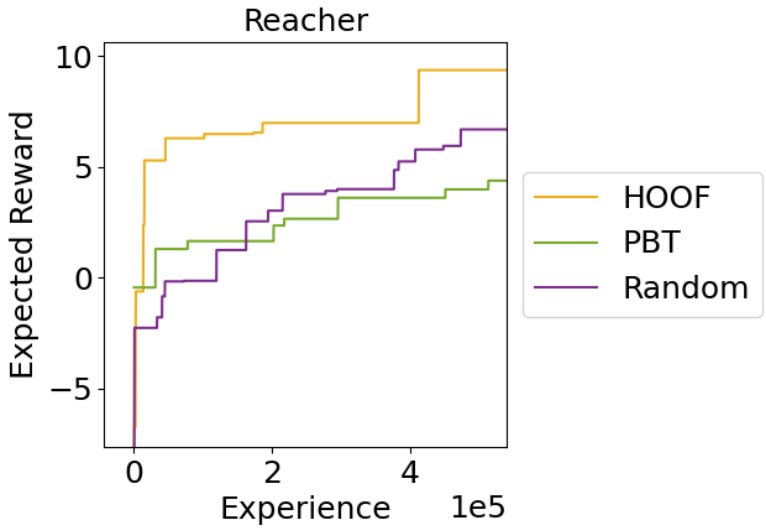

Figure 3.6: Performance of HOOF and PBT compared to random baseline for SAC

As shown above, HOOF outperforms PBT and the random baseline for all of the policy gradient methods. In general, PBT seems to lag behind the random baseline. The poor performance is attributed to the significant amount of experience PBT requires in its training and evaluation process, while the random baseline is a single training run. Generally, PBT would be able to obtain the same or better results as the random baseline but would require significantly more samples.

Hypothesis Two: Performance on Value-Based Methods Since HOOF is not usable for value-based methods, PBT is expected to outperform the random baseline. Due to resource and time constraints, all seeds for the random baseline for DQN on LunarLander were not able to completed. DQN was the most resource intensive RL algorithm in terms of time and resources. So the results for a single seed are shown, these results are more variable but comparable. Figure 3.7 shows the results for DQN on CartPole and LunarLander.

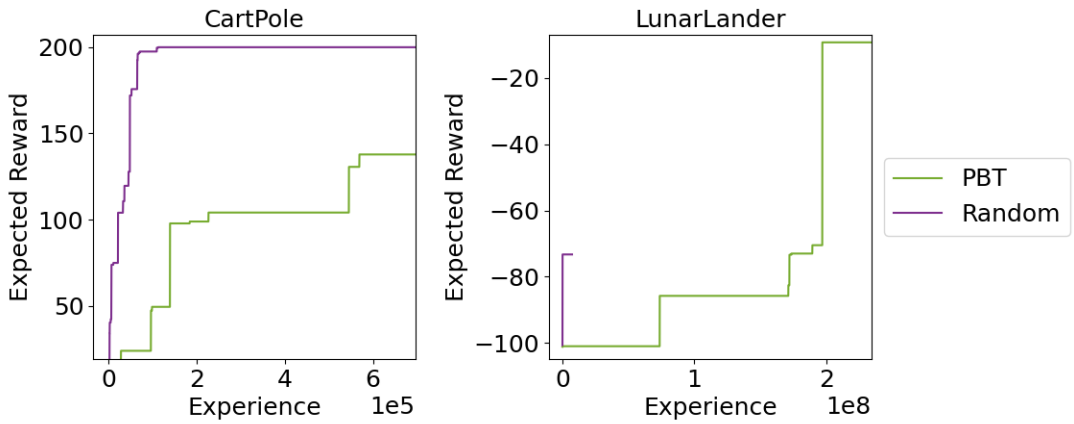

Figure 3.7: Performance of PBT compared to random baseline for DQN

For CartPole, the random baseline actually outperforms PBT. As previously mentioned, the significant amount of samples required for PBT hinders its performance when compared to the random baseline. Since CartPole is an environment with a high fraction of high performing hyperparameter configurations, the exploration required within PBT is a significant hindrance.

The results for PBT compared to the random baseline seem to be misleading due to the number of high performing hyperparameter configurations. In LunarLander, it is expected the random baseline is outperformed by PBT. However, due to the large amount of experience required for this comparison the random baseline was not able to be completed. While a single hyperparameter configuration may outperform PBT if chosen at random, the likelihood of this occurrence is low. Additionally, the time and/or resource required to run several configurations in parallel to improve the chances of identifying a hyperparameter configuration at random would still be slower the PBT. Therefore, PBT is the preferred tuning method due to time and resource contraints.

Summary From the empirical evaluation of this survey of state of the art tuning methodologies, HOOF is the best performing tuning method when using a policy

gradient RL algorithms. PBT would be the best tuning method for value based RL algorithms, even though PBT requires more experience. PBT only requires a single run and is thus seen as the better tuning method even if its performance is delayed for environments that have a high fraction of high performing environments. HOOF does not suffer from this issue since it only requires a single training run with limited experience and dynamically tunes the hyperparameter during that run.

Chapter 4

Hyperparameter Tuning with Bandits and Off-Policy Sampling

This chapter frames the hyperparameter tuning problem within the RL domain as a Multi-Armed Bandit (MAB) problem. To frame this problem as a MAB, I represent each arm as a hyperparameter setting and the feedback is the expected value of the policy. Then, I describe a novel and fully online tuning strategy known as Hyperparameter Tuning with Bandits and Off-Policy Sampling (HT-BOPS). I empirically show that HT-BOPS performs as well as or better than state of the art tuning strategies described in Chapter 3.

4.1 Motivation

While examining the prominent hyperparameter tuning methodologies in current literature, each methodology has its benefits and drawbacks based on how each step of the general hyperparameter tuning method is implemented. A summary of each step for PBT and HOOF can be found in Table 3.1 and 3.2. First, consider the update step which is responsible for generating training data and updating the policies. PBT utilizes Hyperparameter Policy Pair (HPP)s while HOOF holds a single policy. For

PBT training data is generated for each HPP while HOOF generates training data from a single policy. If 10 hyperparameter configurations are used, this results in 10 times the training data for PBT than HOOF. Then, for PBT the policy in each HPP is updated using the associated training data and hyperparameter configuration. HOOF generates candidate policies for each hyperparameter configuration using the base policy and the training data generated from it.

Next is the evaluate step, for PBT evaluation data is generated to calculate the average cumulative reward which is then used to select the HPP. HOOF uses an Off-Policy Evaluation (OPE) method to estimate the reward for each hyperparameter configuration, then this estimate reward is used to select the hyperparameter configuration and associated candidate policy. Again, if 10 hyperparameter configurations are used and 10 evaluation trajectories are generated then this results in 100 trajectories generated to complete the evaluation step for PBT. However, HOOF reuses the training data generated in the update step within the OPE method. Finally, the exploit step for PBT executes two functions which aim to improve policy performance and update hyperparameter configurations. HOOF's update step consists of discard all the candidate policies not selected so a single policy persists across iterations.

PBT is a dynamic solution that works across multiple learning tasks, such as Supervised and Reinforcement Learning. Additionally, it has flexible definitions of the explore and exploit functions and extensively explores each hyperparameter configuration. This extensive exploration results in significant exploration of the solution space but also results in sample inefficiency. PBT requires each configuration to interact with the environment, as well as evaluations to be completed within the environment. While this methodology is extremely flexible, it may be impractical to use for tasks that do not have a environment simulator or in cases where environment interactions are expensive in time and/or cost.

While PBT is sample inefficient, HOOF is sample efficient since it uses the same

samples to generate multiple candidate policies and uses those same samples within the evaluation procedure. However, HOOF selects a new hyperparameter setting using a step-wise greedy strategy. While this methodology addresses the sample inefficiencies of PBT, the step-wise greedy strategy may lead to sub-optimal solutions. Additionally, HOOF only operates on policy gradient or actor-critic methods, which limits its domain of applicability. It is also important to note that PBT does not have this step-wise greedy strategy since it holds multiple policies that are generated using a hyperparameter configuration over multiple training data sets.

Hyperparameter tuning is vital to the success of a RL. Therefore, having a tuning methodology that addresses both of these facets as well as being applicable to both policy gradient and value based methods is critical. HT-BOPS is designed to be sample efficient as well as avoid step-wise greedy selection to prevent sub-optimal solutions, while being applicable to a wide variety of RL algorithms.

4.2 Our Approach

Within the RL domain, data is generated in a cyclic manner where the current model is used to produce data for the next training iteration. This cyclic nature emphasizes the importance of selecting the best hyperparameter configuration at each iteration. Selecting the best hyperparameter configuration at each iteration will facilitates optimal learning and therefore optimal performance. The similarities between the classic MAB problem and this framing of hyperparameter tuning are unmistakable. Therefore, HT-BOPS treats each hyperparameter configuration as an arm. Typically, an arm must be pulled a significant amount of times in order to obtain an estimate. However, a considerable amount of experience within the RL environment would be required to obtain estimates for all arms. In order to reduce this considerable amount of experience, HT-BOPS uses the data collected from the pulled arm to estimate the

value of each of the other arms through the use of OPE. These arm estimates are then used within the arm selection process. HT-BOPS is able to generate a hyperparameter schedule by greedily choosing a hyperparameter configuration from the arm selection process. Within HT-BOPS, a non-stationary bandit solution known as UCB is used.

The next sections describe how HT-BOPS implements the main 3 steps of the general hyperparameter tuning procedure for RL. The first main step within HT-BOPS is the update step which gathers data and updates the policies associated with each hyperparameter configuration and is described in Section 4.2.1. The evaluate step is described in Section 4.2.2 and is responsible for evaluating each arm and then utilizes the estimate of each arm, or hyperparameter configuration, within the arm selection process in order to select the next configuration. The selected hyperparameter configuration is added to the history and the behavioral policy for the next training iteration is set. The final step, referred to as the exploit step exploits high performing policies in order to improve performance of under performing policies in later training iterations. This step is described in Section 4.2.3.

4.2.1 Update

HT-BOPS's update step is described in Algorithm 11. The currently selected hyperparameter configuration, ϕ_i, has an associated policy, π_i, for each iteration of the tuning procedure. Training data is gathered on line 1 by taking the initial state from the environment and sampling an action using the probabilistic policy, π_i. This is repeated for a fixed number of steps to generate a batch of steps or continued till a terminal step is reach a full trajectory. It is important to note that since the original policy is the same across hyperparameter configurations, any policy may be chosen initially.

Algorithm 11 HT-BOPS Update Subroutine

Input: Selected policy, π_i, and Hyperparameter-Policy Pair Set, HPP
Output: Hyperparameter-Policy Pair Set, HPP, and training data, τ
1: Sample training data τ using π_i
2: **for** $(\phi_n, \pi_n) \in HPP$ **do**
3: $\quad \pi'_n \leftarrow$ Update π_n using ϕ_n and τ
4: $\quad HPP_n \leftarrow (\phi_n, \pi'_n)$
5: **end for**
6: **return** HPP, τ

Each HPP is iterated through and the policy is updated using the hyperparameter configuration and the training data on line 3. Each of the HPP independently updates its held policy which can result in a diverse set of policies. This update is defined by the learning algorithm as was further described in Section 2.3. For example, if the RL algorithm that is being used is DQN then value function approximation will be updated using the following rule:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \Big(\frac{1}{2} \big(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a) \big)^2 \Big) \tag{4.1}$$

where s is the state, a is the action, r is the reward, s' is the next state, γ is the discount factor, α is the learning rate and Q_θ is the function approximator parameterized by θ. Since DQN is a value based method the update rule applies to the underlying value function that derives the policy.

4.2.2 Evaluate

As previously mentioned, an arm must be pulled a significant amount of times in order to obtain an estimate. HT-BOPS avoids this additional experience by obtaining estimates for all the arms through the use of Off-Policy Evaluation (OPE).

Off-Policy Evaluation Two OPE methodologies used within HT-BOPS: Weighted Importance Sampling (WIS) and Sampling Importance Resampling-Particle Filter

(SIR-PF). The first methodology is WIS and was originally described in Section 2.4.1. WIS is defined as follows:

$$WIS(\pi_b, \pi_e, H) = \frac{\sum_{\tau}^{H} IW(\pi_b, \pi_e, \tau) R_\tau}{\sum_{\tau}^{H} IW(\pi_b, \pi_e, \tau)} \tag{4.2}$$

where

$$IW(\pi_b, \pi_e, \tau) = \frac{\prod_{i=0}^{T} \pi_e(a_i|s_i)}{\prod_{i=0}^{T} \pi_b(a_i|s_i)} \tag{4.3}$$

where π_b is the behavior policy and π_e is the evaluation policy and τ is the data that can be a batch of steps or a full trajectory. The behavior policy is the policy that was used to generate the history while the evaluation policy is the policy that is being evaluated in comparison to the behavior policy.

Importance Sampling (IS) and its variants are typically used to estimate a stationary distribution based on a history of samples from another stationary distribution. In HT-BOPS, the evaluation trajectory history is not generated from a single stationary distribution, i.e. policy, but rather multiple non-stationary distributions. However, WIS can be used within the first tuning iteration since the data is generated by a single stationary distribution. WIS provides a sufficient initial estimate since initial estimates are generally poor. Thus, HT-BOPS uses WIS for the first tuning iteration.

However, any subsequent tuning iteration that uses the evaluation trajectory history must be able to handle data generated by multiple non-stationary distributions. The multiple non-stationary distributions is an artifact of the selection of hyperparameter configurations and their corresponding policies changing throughout training. Even if a single policy is selected multiple times in a row, the policy has been updated and therefore the distribution has changed. This results in a history of evaluation trajectories that are generated from multiple non-stationary distributions.

Due to this, a different solution is required. Thus, HT-BOPS uses a novel approach that utilizes a SIR-PF to construct an off-policy estimate of the hyperparam-

eter configuration value [20]. Particle filtering uses a set of particles, or samples, to represent the posterior distribution of some stochastic process given noisy or partial observations [15]. SIR-PF expands a classic particle filter by re-sampling particles in order to remove highly unlikely particles and obtain new, more likely particles. A particle is defined as a trajectory within the SIR-PF. The SIR-PF requires a history of trajectories which consists of all the steps that make up the trajectory. A step within a trajectory consists of the state, action and reward. The behavior policy is the policy that was used to generate the trajectory and the action probability is the probability of the action under the behavioral policy. Since the history consists of multiple trajectories, there is likely to be a behavioral policy for each trajectory. In addition to the history, the evaluation policy and the number of samples to be taken is required. The evaluation policy is the policy that whose value is being estimated.

In the original formulation of the SIR-PF, the distribution that is sampled from is defined as the probability of the particle and the reward as the importance weighted discounted reward. The probability of a particle, in this case, is defined as:

$$p(\tau) = \prod_{i}^{L} \pi(a_i|s_i) \tag{4.4}$$

where τ is the trajectory and s_i, a_i are the state-action pairs for each step. However, this formulation would favor shorter trajectories since $p(\tau)$ approaches 0 as L increases. However, the shorter trajectory preference would be counter productive for maintenance tasks. A maintenance task is a task that aims to control variables to remain within a specific range. For example, the CartPole environment aims to control the cart such that the pole remains upright and thus longer trajectories are preferred. Another concern with this formulation is that the importance weighted reward is unbounded. An IW is defined as the ratio of the products of probabilities which are guaranteed to be bounded $[0, 1]$. However, a single action probability

of 0 makes the importance weight undefined and furthermore any near zero action probability can result in large IW. These widely varying importance weights are then multiplied by the reward resulting in an unbounded reward estimate.

To mitigate this concern, the IW are normalized to form a probability distribution. The IW calculation utilizes the probabilities and the relative impact while not favoring short trajectories due to their relative nature. Originally, the importance weighted discounted rewards would be used, yet due to the discussed change, it seems fitting to use the discounted rewards. However, using the discounted rewards induces a bias for larger discount factors. Larger discount factors produce larger rewards since they look more into the future of a trajectory. For these reasons, we defined the sampling distribution as the normalized importance weights and the reward as the cumulative un-discounted reward as defined the environment. The resultant procedure for the SIR-PF is defined in Algorithm 12.

Algorithm 12 SIR PF

Input: Evaluation trajectory history, W, behavior policy, π_b, and evaluation policy, π_e

Output: Expected value of policy and sample variance

1: Initialize importance weight, w and reward, v histories
2: **for** $i = 0, ..., len(W)$ **do**
3: $\quad w_i \leftarrow IW(\pi_b, \pi_e, W_i)$ using Eq. 4.3
4: $\quad v_i \leftarrow \sum r \ \forall \ (s, a, r) \in W_i$
5: **end for**
6: $w \leftarrow \frac{w_i}{\sum_j w_j} \ \forall \ i \in len(w)$
7: Bootstrap sample from v using w
8: Compute $\hat{r}, \sigma_r$ of sampled set
9: **return** $\hat{r}, \sigma_r$

For each particle, or trajectory, the IW and cumulative reward is calculated on lines 2-5. The IW values are normalized to form a probability distribution on line 6. This probability distribution is used to sample from the cumulative reward as shown on line 7. Then, the mean and variance of the sampled set is returned to be used in the arm selection process.

Non-Stationary Bandits The non-stationary bandit problem assumes that the distribution associated with the reward of each arm may vary across time. In literature, there are commonly two settings for how the reward distributions change across time: piece-wise and slowly varying [6]. The piece-wise setting defines the distribution fixed for a period of time and then abruptly changes at some number of points. The slowly varying setting assumes that the reward distribution is gradually changing over time.

One solution to the non-stationary bandit problem within the UCB family is known as Sliding-Window Upper Confidence Bound (SWUCB) [6]. While UCB1 accumulates use counts of each arm throughout the entire history, SWUCB accumulates use counts of each arm throughout the sliding window. The sliding window is a fixed width view of the most recent trajectories added to the evaluation trajectory history. This windowed view results in the confidence term as follows:

$$U_t(a) = \sqrt{\frac{2(c_2 - c_1)^2 \log(\min(t, \tau))}{N_t(a)}} \tag{4.5}$$

where τ is the window size, $N_t(a)$ is the number of times arm a has been chosen at time t, and $[c_1, c_2]$ are the reward bounds as defined by the environment. In the original UCB1 formaultion, as noted in Section 2.2.4, the probability bound was chosen to be t^{-4}. This choice is also reflected in the SWUCB algorithm. However, this choice was initially chosen due to an infinite sum, since the history was assumed to infinitely grow over time. Since the history is now finite, due to the sliding window, this choice is no longer necessary. HT-BOPS uses a fixed choice of p so the variance term does not overwhelm the calculation. For example, if the window size is 60, then the probability of the estimate being outside the range would be $7.7e - 8$. This high confidence results in very high variances that overwhelm the estimate. Thus, fixing the choice of p to be, for example 0.01, reduces the variance term while remaining

highly confident in the estimate. The choice of p can now be set as a property of the environment and/or algorithm choice.

It is important to note that the choice of using bounded rewards within SIR-PF turns out to be vital since the unbounded reward generated from importance weighted rewards would not satisfy the conditions of the UCB algorithm. The UCB score is adopted to incorporate two types of variance we expect to encounter within the hyperparameter tuning context: the variance of the choice of arms, UCB uncertainty term, as well as the variance of the reward estimate, variance of the OPE. Thus, the expanded formulation to include the variance of the reward estimate is as follows:

$$UCB^*(\pi_b, \pi_e, W) = \mu(\pi_b, \pi_e, W)_{PF} + \sqrt{\frac{(b-a)^2 \log(p \min(t, \tau))}{-2N_t(a)} + \sigma(\pi_b, \pi_e, W)_{PF}} \tag{4.6}$$

where μ_{PF} and σ_{PF} are the estimated reward and sample variance from the SIR-PF, τ is the window size, $N_t(a)$ is the number of times arm a has been chosen at time t, $[c_1, c_2]$ are the reward bounds as defined by the environment and p is the bounding probability. The variances the above formulation are able to be combined without scaling since they are both sample variances, one from arm usage and the other from the SIR-PF, of the reward and thus are comparable.

HT-BOPS evaluation step is described in Algorithm 13. The procedure for the first training iteration is described by lines 2-3. This portion of the procedure uses WIS on the training data generated in the update step for the OPE. The estimates are produced on line 2, then all of the use counts within UCB are set to 1 on line 3. WIS was chosen since initial estimates are generally poor and the training data is generated under a single, fixed distribution. This initial WIS estimate provides enough info for early selection and allows the algorithm to avoid the extra arms pulls to generate an initial estimate for each hyperparameter configuration.

Algorithm 13 HT-BOPS Evaluation Subroutine

Input: Training iteration, i, training data, τ_t, selected policy, π_i, evaluation trajectory window, W, and Hyperparameter-Policy Pair Set, HPP

Output: Policy π, Hyperparameter Schedule H, evaluation trajectory window, W

1: **if** $i == 0$ **then**
2: $V \leftarrow \{WIS(\pi_i, \pi_n, \tau_t)\ \forall\ (\phi_n, \pi_n) \in HPP\}$ using Eq 4.2
3: Set use count for each configuration within UCB to 1
4: **else**
5: Sample evaluation trajectory, τ_e, using π_i
6: $W \leftarrow W \cup \tau_e$
7: $V \leftarrow \{UCB^*(\pi_i, \pi_n, W)\ \forall\ (\phi_n, \pi_n) \in HPP\}$ using Alg. 12 and Eq. 4.6
8: **end if**
9: $n = \underset{n}{\text{argmax}}\, V_n$
10: **return** V, π_n, ϕ_n

After the first iteration, the UCB procedure, as described in Equation 4.6, is estimate the value of each arm, or hyperparameter configuration. This procedure is outlined in lines 5-7. The currently selected hyperparameter configuration, ϕ_i, has an associated policy, π_i, for each iteration of the tuning procedure. An evaluation trajectory is generated using π_i on line 5. This evaluation trajectory is then added to the evaluation trajectory window, W, on line 6. It is important to note that the history has a fixed size, M. When the history has reached its size limit, the early trajectories are replaced with the new trajectories. Then the evaluation trajectory history is used to generate the arm value estimates on line 7 by using Equation 4.6. Finally, line 9 greedily selects the arm, which consists of the hyperparameter configuration and corresponding policy, from these estimates.

4.2.3 Exploit

Having each hyperparameter configuration independently update a policy encourages exploration within the solution space. However, this exploration can lead to a policy being stuck in a sub-optimal region within the solution space. For example, consider two hyperparameter configurations, one of which is good during early phases

of learning with the other is good during later phases. When the first hyperparameter configuration performance starts to decline, the policy associated with the second may be lagging behind since it was updated using a sub-optimal hyperparameter configuration. Thus, the optimal schedule may not be identified or may take significantly more experience to achieve.

The exploitation step within HT-BOPS aims to mitigate this issue by copying the policy from top performing hyperparameter configurations to under performing hyperparameter configurations. HT-BOPS exploit step is described in Algorithm 14. A counter, shown on line 1, records the number of training iterations since the exploit procedure has been completed. Once the minimum number of training iterations were completed, the exploit step is executed as described in lines 3-5. Based on the arm estimates, generated in evaluation step of HT-BOPS, the top and bottom k performing hyperparameter configurations are identified in line 3. Then line 4 shows the replacement of the bottom k policies with the top k policies. The choice of which top k policies is chosen through uniform random sampling. Finally, line 5 resets the training iteration counter so that a minimum number of training iterations are completed before performing another exploit step.

Algorithm 14 HT-BOPS Exploit Subroutine

Input: Training iteration counter, t_e, minimum training iterations, E, hyperparameter configuration value estimates, V, and Hyperparameter-Policy Pair Set, HPP
Output: Hyperparameter-Policy Pair Set, HPP, and training iteration counter, t_e

1: $t_e \leftarrow t_e + 1$
2: **if** $t_e > E$ **then**
3: Identify top and bottom k performing configurations using V
4: Replace bottom k% policies by sampling top k policies
5: $t_e \leftarrow 0$
6: **end if**
7: **return** HPP, t_e

It is important to note that this step is similar to a common implementation of the exploit step for PBT. A common implementation of the exploit step is to

identify the top and bottom k performing hyperparameter settings based on the value estimate from the exploit trajectories used in PBT. However, in HT-BOPS, the same strategy is utilizes but based on the computed arm estimates. Only the most under performing hyperparameter configurations are chosen to be exploited so the majority of configurations continue to explore the solution space.

4.2.4 Hyper-Hyperparameters

HT-BOPS has hyperparameters of its own, known as hyper-hyperparameters. Since a search space, number of training iterations, and number of hyperparameter configurations must be defined for HT-BOPS, they are not necessarily considered hyperparameters because they are common across all tuning methodologies. The three hyperparameters for HT-BOPS are as follows:

- Minimum Training Iterations before Policy Exploitation
- Window Size for UCB
- Bounding Probability for UCB

Additionally, a number of configurations to exploit is noted within Algorithm 14. However, this value was fixed to 20% of the total hyperparameter configurations considered for a tuning method for all experiments. This value was chosen due to its popular use within PBT exploit method implementations. However, this value may be altered if additional exploitation would be required.

4.3 Approach Comparison

HT-BOPS frames the hyperparameter tuning problem, within the RL setting, differently from any method in current literature. In this section, the similarities and differences between HT-BOPS and two of the state of the art approaches, discussed

in Chapter 3, are highlighted. This comparison is broken down across the three main steps of HT-BOPS. The summary of how HT-BOPS implements each of the steps is shown in Table 4.1. In general, HT-BOPS and PBT both associated a policy and hyperparameter configuration which is referred to as a Hyperparameter Policy Pair (HPP).

Step	HT-BOPS
Update	• Generate training data from selected HPP • Update the policy using training data and hyperparameters for each HPP
Evaluate	• Off-Policy Evaluation- Weighted Importance Sampling or Sampling Importance Resample Particle Filter • Selects hyperparameter configuration with maximum Upper Confidence Bound value
Exploit	• Copy policy values from top performing policies to under performing policies

Table 4.1: HT-BOPS Step Definitions

4.3.1 Update

The first step of the general tuning procedure of RL is the update step which gathers training data and updates the policies using the RL algorithm's update rule. Table 4.2 describes each update step of the tuning strategies used within the comparisons in this work. The update step can be seen as broken into two functions: how the training data is generated and how the update is completed. HT-BOPS and HOOF both generate training data from a single policy while PBT generates training data for every policy. PBT then updates each HPP with the associated training data while HT-BOPS updates each HPP with the training data generated from the single policy. HOOF generates candidate policies by pairing the initial policy and generated training data with each hyperparameter configuration.

Tuning Method	Update
HT-BOPS	• Generate training data from selected HPP • Update the policy using training data and hyperparameters for each HPP
HOOF	• Generate training data from the policy • Generate candidate policies for each hyperparameter configuration
PBT	• Generate training data from each policy for all HPPs • The policy in each HPP is updated using the associated training data and hyperparameter configuration

Table 4.2: Tuning Method Update Step Definitions

PBT requires significantly more training data to be acquired. HT-BOPS maximizes the use of the information from the training data gathered from a single hyperparameter configuration and corresponding policy by using it across all of the hyperparameter configurations and their corresponding policies. This reuse reduces the requirement of data to be generated per iteration. HOOF and HT-BOPS generate the same amount of training data but the difference between the method is focused on the difference in a single policy being held versus a collection of HPPs.

4.3.2 Evaluate

The second step is the evaluate step which is responsible for evaluate the hyperparameter configurations and selects a single configuration. This step can be split into two steps, the first of which produces an estimate of each hyperparameter configuration's value, or expected reward, then the second is how the next hyperparameter configuration is chosen. The summary of this step across tuning method is shown in Table 4.3. HOOF and HT-BOPS both use OPE methods to estimate the reward of each hyperparameter configuration. HOOF uses training data within the OPE method while HT-BOPS uses evaluation data since evaluation data generates a stronger value signal of a policy compared to training data. Generating an evaluation trajectory at each

iteration increases sample complexity, however, the evaluation trajectories are more representative of the policies behavior. HOOF uses WIS as its OPE method, while HT-BOPS uses WIS for the first iteration then uses SIR-PF.

Tuning Method	Evaluate
HT-BOPS	• Off-Policy Evaluation-Weighted Importance Sampling or Sampling Importance Resample Particle Filter • Selects hyperparameter configuration with maximum Upper Confidence Bound value
HOOF	• Off-Policy Evaluation-Weighted Importance Sampling • Selects hyperparameter configuration with maximum estimated reward
PBT	• Generates evaluation trajectories and calculate average cumulative reward • Selects HPP with maximum reward

Table 4.3: Tuning Method Evaluate Step Definitions

PBT generates evaluation trajectories and then records the average cumulative reward for each hyperparameter configuration. PBT and HOOF selects the hyperparameter configuration with maximum reward or estimate reward, respectively. However, HT-BOPS additionally calculates the UCB value for each hyperparameter configuration. This value includes two sample variance terms that represent the variance of the samples based on the hyperparameter use count and the variance associated with the OPE method. HT-BOPS selects a hyperparameter configuration based on the UCB values.

4.3.3 Exploit

The third and final step is the exploit step which employs a strategy to take advantages of well performing hyperparameters. The summary of each tuning method's implementation of the exploit step is shown in Table 4.4. PBT executes its two custom functions in order to improve the policy performance and get new or update

existing hyperparameter configurations. HT-BOPS copies policies of high performing HPPs to low performing HPPs. It is important to note that PBT's custom functions can be written to match the behavior of HT-BOPS's exploit step. However, the similarity of HT-BOPS and PBT is dependent on the implementation details. Finally, HOOF discards all candidate policies not selected.

Tuning Method	Exploit
HT-BOPS	• Copy policy values from top performing policies to under performing policies
HOOF	• Discards all candidate policies not selected
PBT	• Execute exploit function is used to improve policy performance • Execute explore function is used to get new or update existing hyperparameter configurations

Table 4.4: Tuning Method Evaluate Step Definitions

4.4 Summary

HT-BOPS is an extremely flexible tuning method due to its ability to handle discrete and continuous action and state spaces as well as a variety of RL algorithms. However, the two OPE methods utilized in HT-BOPS assume the use of a stochastic policy where an action has a probability. It is important to note that some RL algorithm operate strictly on deterministic policies, such as Deep Deterministic Policy Gradient (DDPG), that map a state directly to an action without a probability distribution [12]. However, this family of RL algorithms construct the policy based on a value estimate. This action value estimate can be used within the OPE instead of the action probability. It has been previously shown that algorithms, such as REINFORCE, translate estimated action values into probabilities [27]. A supplemental function may be added to convert action values into probabilities to be used within the OPE. Since the probabilities are only used within the Importance Weight (IW) calculation

and are therefore relative, the action values may be used in place of the probabilities as long as they are strictly positive.

Additionally, some concerns about HT-BOPS exist. As noted in Section 2.4.1, Importance Sampling (IS) suffers from high variance as the evaluation policy and behavior policy diverge. This can causes inconsistencies within the IW which result in potentially inconsistent results from the SIR-PF. Furthermore, HT-BOPS focuses on exploration of the solution space which could hinder its performance on easily solved environments.

Chapter 5

Empirical Evaluation

In this chapter, I compare tuning strategies across environments and learning algorithms as well as evaluate regret of selected hyperparameters for a tuning method. These experiments will be performed using the same experimental methodology as described in Section 3.8. The tuning methods used within this evaluation, Hyperparameter Optimization On the Fly (HOOF) and Population Based Training (PBT), are described in Chapter 3, while the presented method, Hyperparameter Tuning with Bandits and Off-Policy Sampling (HT-BOPS), is described in Chapter 4. Additionally, descriptions of all the RL algorithms used within this section can be found in Section 2.3. HOOF will be used to compare against HT-BOPS for policy gradient methods: PPO, A2C, and SAC while PBT will be used to compare against for the value-based method: DQN.

5.1 Environment and Learning Algorithm Evaluation

This set of hypotheses focus on comparing tuning strategies across a variety of environments and RL learning algorithms. This section will present the results to test

the following hypotheses:

1. HT-BOPS outperforms HOOF on average for policy gradient and actor-critic algorithms.
2. HT-BOPS outperforms PBT on average for value function-based algorithms.
3. A2C produces best results with HOOF, while all of the other algorithms perform better with HT-BOPS.
4. HOOF outperforms HT-BOPS on environments with numerous high performing hyperparameter configurations.
5. HT-BOPS outperforms HOOF on environments with scarce high performing hyperparameter configurations.
6. A hyperparameter schedule outperforms a fixed hyperparameter configuration on average.

As previously mentioned in Section 3.8.2, a tuning method can outperform another in two ways: asymptotic reward and convergence rate. The asymptotic reward is defined as the policy with largest average cumulative reward found by a tuning method which would presumably be the maximum reward of the environment if enough training experience is gathered. The convergence rate is how fast the tuning method is able to learn the policy that has the maximum reward of the environment. Both of these results are reported on average for the experiments presented within this thesis. Additionally, the hyperparameter configurations across tuning methods remain consistent and the configurations are generated using Latin Hypercube Sampling (LHS).

5.1.1 Hypothesis One: Policy Gradient and Actor-Critic Performance

The best performing tuning method for policy gradient methods in Section 3.8 was HOOF, so it will be used as the comparison to HT-BOPS for policy gradient methods. The results of the first policy gradient algorithm, A2C, are visualized in Figure 5.1. For A2C, HOOF and HT-BOPS have the same asymptotic reward for CartPole, while HOOF is able to achieve a larger asymptotic reward for LunarLander. The reasoning for this performance wil be discussed in Section 5.1.3. On Reacher, the opposite is found, where HT-BOPS achieves a larger asymptotic reward than HOOF.

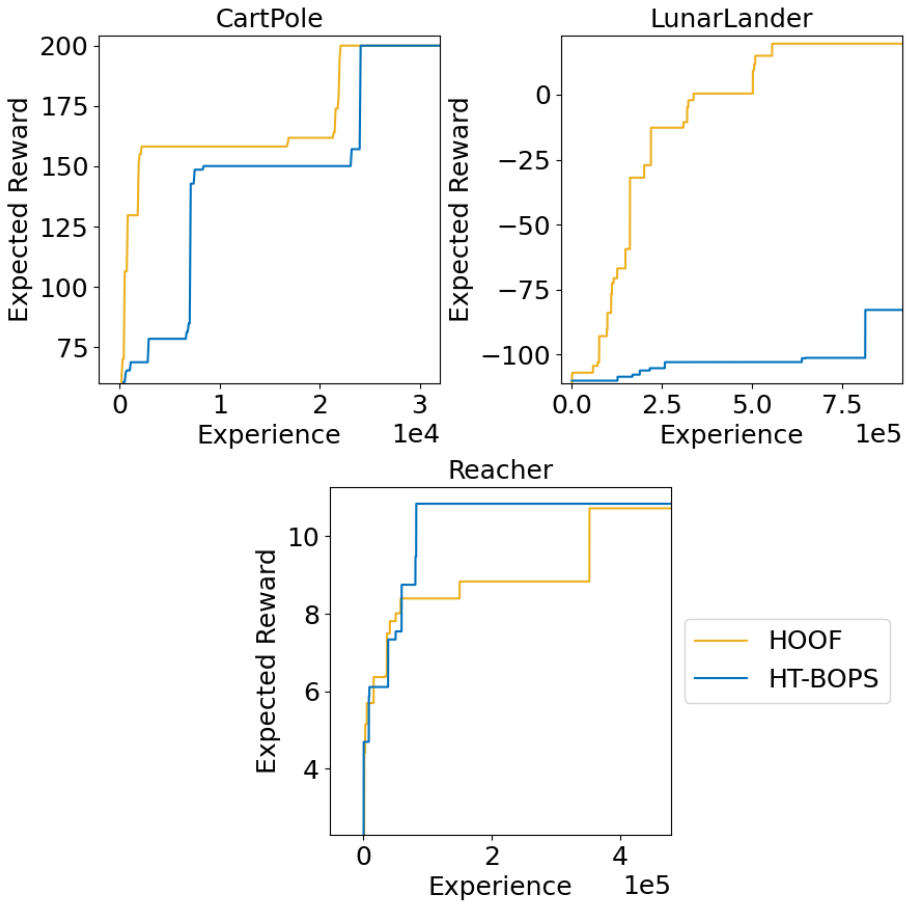

Figure 5.1: Performance of HT-BOPS compared to HOOF for A2C

Table 5.1 summarizes the convergence rate at various thresholds for A2C, which shows the experience required to achieve various thresholds of the environments maximum reward. In order to compare convergence rate, various thresholds were set to represent different phases of learning. The minimum experience, across tuning methods, to achieve that threshold of the maximum reward is highlighted in blue for each row. A2C on LunarLander is never able to achieve the minimum fraction of the maximum reward, which, as noted previously, will be discussed in Section 5.1.3. The performance across HOOF and HT-BOPS is comparable for CartPole in the first phase of learning. HOOF performs better in the middle phases of learning, since it as able to convergence faster to 50% and 75% of the maximum reward. However, the convergence rates for the final stage of learning is again comparable. A similar trend occurs for Reacher, however, the initial and final learning phases show higher convergence rates for HT-BOPS.

Environment	Tuning Algorithm	Experience for N% of Maximum Reward			
		25%	50%	75%	90%
CartPole	HOOF	200	500	1,090	21,900
	HT-BOPS	100	7,100	7,500	24,100
LunarLander	HOOF	1.09e5	1.61e5	2.19e5	5.03e5
	HT-BOPS	N/A	N/A	N/A	N/A
Reacher	HOOF	1,600	3,100	41,500	3.52e5
	HT-BOPS	700	8,800	50,500	81,300

Table 5.1: Experience to achieve various thresholds of maximum reward for A2C

The next policy gradient algorithm to be discussed is PPO and the asymptotic reward results are shown in Figure 5.2. HT-BOPS and HOOF have achieved the same asymptotic reward on CartPole while HT-BOPS is able to achieve a larger reward than HOOF on the other two environments. Next, the convergence rate at various thresholds for PPO is summarized in Table 5.2. Again, HT-BOPS and HOOF convergence rate for the first phase of learning are comparable, but for every additional phase of learning, HOOF has a faster rate of convergence. Performance of PPO on

LunarLander and Reacher have a similar pattern, the first phase of learning are comparable or HOOF has a higher rate of convergence, respectively. The middle phases of learning result in HOOF having a higher convergence rate for both environments, while the convergence rate for the last phase of learning is highest for HT-BOPS.

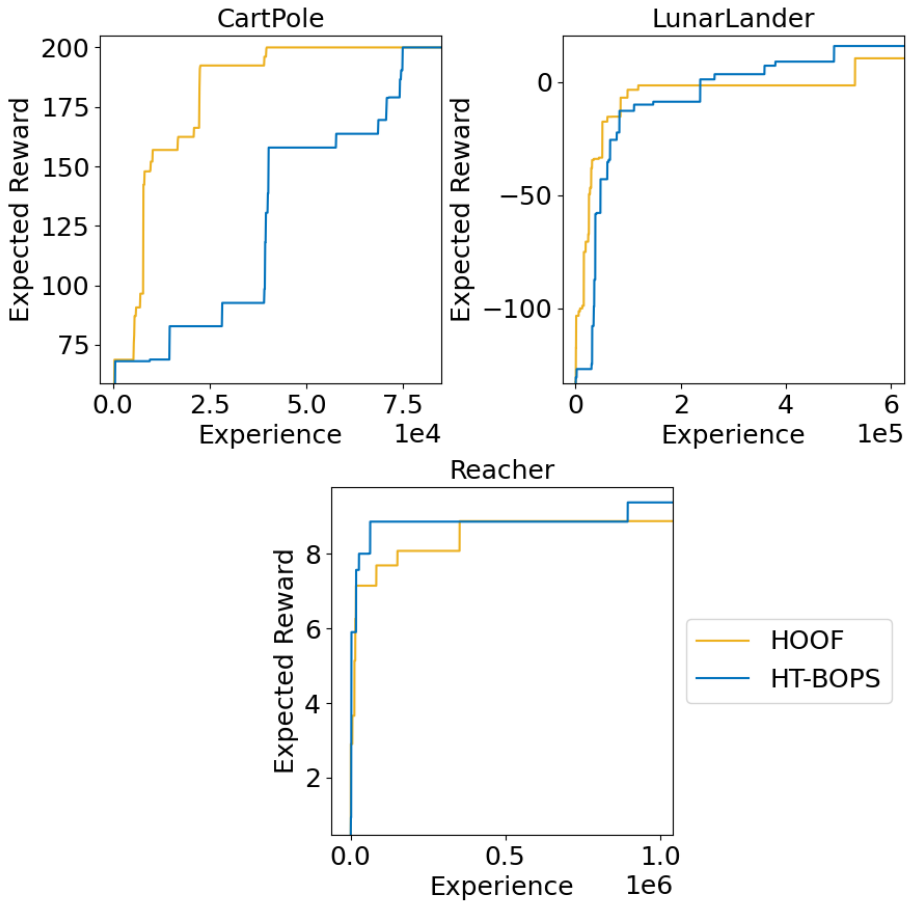

Figure 5.2: Performance of HT-BOPS compared to HOOF for PPO

Environment	Tuning Algorithm	Experience for N% of Maximum Reward 25%	50%	75%	90%
CartPole	HOOF	100	7,900	8,200	22,400
	HT-BOPS	100	39,200	40,300	70,900
LunarLander	HOOF	15,700	25,400	51,000	5.31e5
	HT-BOPS	35,500	37,500	83,400	3.60e5
Reacher	HOOF	300	600	15,200	3.52e5
	HT-BOPS	300	2,300	17,700	63,100

Table 5.2: Experience to achieve various thresholds of maximum reward for PPO

The last policy gradient method used within these experiments was SAC and its asymptotic reward results are shown in Figure 5.3. While the performance of HOOF and HT-BOPS are similar, HT-BOPS is able to achieve a larger reward. Additionally, this again illustrates that HOOF can quickly identify a local maximum but again remains there for a significant amount of experience, while HT-BOPS continues to improve while additional experience is gathered.

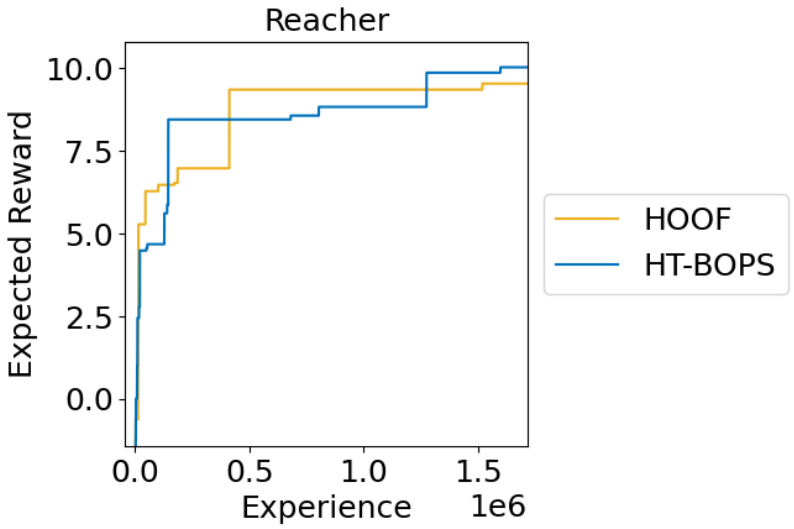

Figure 5.3: Performance of HT-BOPS compared to HOOF for SAC

The convergence rates for SAC is summarized in Table 5.3. Similar to previous experiments, HOOF achieves a higher convergence rate during early learning while

HT-BOPS achieves a higher convergence rate during later learning. However, in this experiment, HOOF is able to also convergence faster for the last phase of learning. When further examining, Figure 5.3, between the experience range of $2.5e5$ and $5e5$, HOOF seems to plateau earlier than in previous experiments. SAC is different from the other policy gradient methods discussed because it is an off-policy method which facilitates continuous exploration [27]. The RL algorithms continuous exploration is likely hindering HOOF's early learning convergence rates.

Environment	Tuning Algorithm	Experience for N% of Maximum Reward			
		25%	50%	75%	90%
Reacher	HOOF	400	400	13,500	1.86e5
	HT-BOPS	4,200	21,200	1.45e5	2.03e5

Table 5.3: Experience to achieve various thresholds of maximum reward for SAC

While HOOF outperforms HT-BOPS in some specific cases, on average, HT-BOPS outperforms HOOF with respect to asymptotic reward. Additionally, HT-BOPS can have comparable or better early learning phase convergence rates while having consistently higher convergence rates in later phases of learning. Therefore, HT-BOPS should be used as the preferred tuning method for policy gradient methods with the exception of some cases specific cases.

5.1.2 Hypothesis Two: Value-Based Performance

The best performing tuning method for value-based methods in Section 3.8 was PBT, so it will be used as the comparison to HT-BOPS for value-based methods. While only one value-based method was used in these experiments, the asymptotic reward results for DQN are visualized in Figure 5.4.

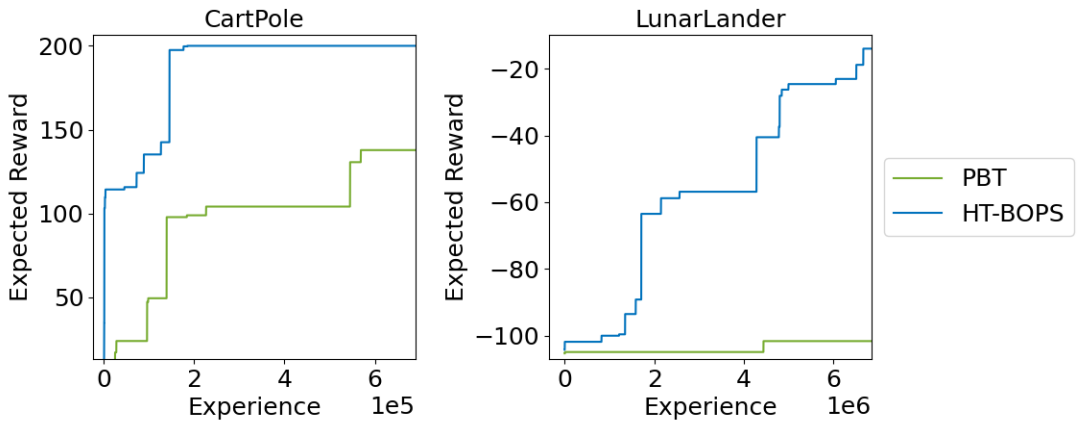

Figure 5.4: Performance of HT-BOPS compared to PBT for DQN

HT-BOPS is able to achieve larger asymptotic rewards on both environments compared to PBT. Additionally, Table 5.4 summarizes the converges rates for various thresholds. Again, HT-BOPS has faster convergence during every phase of learning compared to PBT.

Environment	Tuning Algorithm	Experience for N% of Maximum Reward			
		25%	50%	75%	90%
CartPole	PBT	98,000	1.39e5	7.75e5	2.05e6
	HT-BOPS	900	90,000	1.45e5	1.45e5
LunarLander	PBT	1.51e8	1.59e8	1.77e8	N/A
	HT-BOPS	1.71e6	4.28e6	6.87e6	N/A

Table 5.4: Experience to achieve various thresholds of maximum reward for DQN

These results show that HT-BOPS significantly outperforms PBT for each environment used in these experiments with respect to asymptotic reward and convergence rate. The significant improvement over PBT is based on the large amount of experience required by PBT to train and evaluate each individual hyperparameter configuration. Therefore, HT-BOPS should be used as the preferred tuning method for value-based methods.

5.1.3 Hypothesis Three: A2C Performance

HT-BOPS seemed to struggle with A2C on the LunarLander environment compared to HOOF. To further understand this sub-optimal performance, A2C needs to be further understood. A2C is a first order method since it uses the first derivative/gradient while some methods are considered second order if they use the second derivative/gradient. IW are used within the both formulations of the OPE method within HT-BOPS. A concern for HT-BOPS focused on policies diverging which cause IW to have high variance which can cause HT-BOPS to misbehave. Since A2C is a first order method and does not restrict policies from diverging, this sub-optimal performance is likely HT-BOPS misbehaving due to inconsistent IW. If policy divergence is restricted, performance would improve. To confirm this claim, the gradient of A2C was clipped, in a similar fashion to PPO, internal to the RL learning algorithm. The additional results are shown alongside the original results in Figure 5.5, which shows a significant improvement in performance for HT-BOPS.

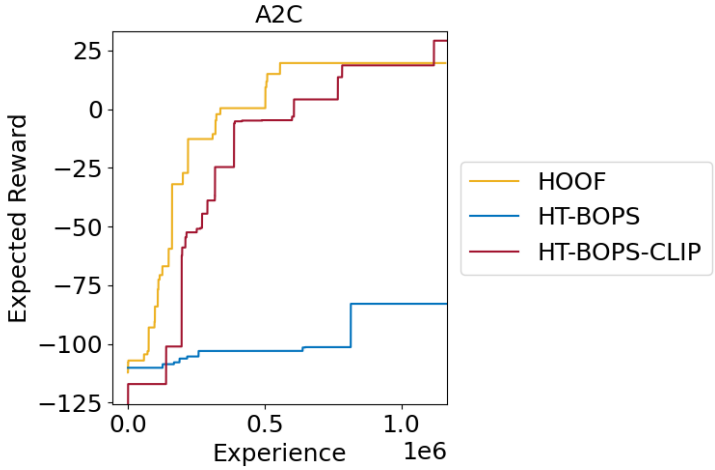

Figure 5.5: Performance of HT-BOPS with internally clipped gradient compared to HT-BOPS and HOOF on LunarLander with A2C

HOOF varies its procedure depending on the order the underlying algorithm.

If the algorithm is a first order method, HOOF restricts the update such that the new policy is within KL distance from the original policy. Any hyperparameter configurations that do not satisfy this constraint are not able to be selected. Large gradient signals would cause a significant policy change, which would result in the OPE estimator used by HOOF to have high variance. In order to maintain meaningful results, HOOF introduced this KL constraint. This constraint creates an artificial restriction to the RL learning algorithm in order to control policy divergence. The external restriction of HOOF is likely the reason why there is a significant performance difference for A2C.

An additional explanation for this behavior may be how the policies are being updated in HT-BOPS. Data generated by a single policy is updating all policies held by HT-BOPS, thus the policy used to sample the training data and the policy being updated is different for all policies but one. This update method is more similar to that of offline RL. An action with low probability under the evaluation policy could force significant policy changes. This misbehavior can be corrected for by modifying the update rule for offline RL [11]. However, with the existing procedure this misbehaviors are minimized by starting with the same initial policy, the exploit step of HT-BOPS and through use of RL algorithms that handle policy divergence.

Some environments and/or RL algorithms can result in large changes in policies which can lead to sub-optimal behavior and delayed convergence. Since the clipped version of A2C was able to achieve a higher maximum asymptotic reward, it is suggested that RL algorithm that handle large policy changes be used for all RL tasks. In fact, most state of the art RL algorithms handle these large policy changes in a variety of ways. However, HOOF is the preferred tuning method when an RL algorithm that does not handle large policy changes is used such asA2C. However, if a RL algorithm that handles large policy changes is used, HT-BOPS is preferred.

5.1.4 Hypothesis Four: Numerous High Performing Configurations

Complexity of environments can vary greatly, so the environments used within these experiments was categorized in Section 3.8.1. An environment with a high fraction of high performing hyperparameter configurations is less sensitive to the choice of hyperparameter configurations at each time step. Only CartPole was categorized as having a high fraction. First, asymptotic reward is examined within Figure 5.6 for all algorithms on Cartpole.

HOOF and HT-BOPS reach the maximum asymptotic reward for this environment for every RL algorithm. PBT was the only tuning method to not reach the maximum reward within experience limit. Thus, HOOF and HT-BOPS have equivalent performance while HT-BOPS outperforms PBT. The second metric for performance to be discussed is convergence rate which is summarized in Table 5.5.

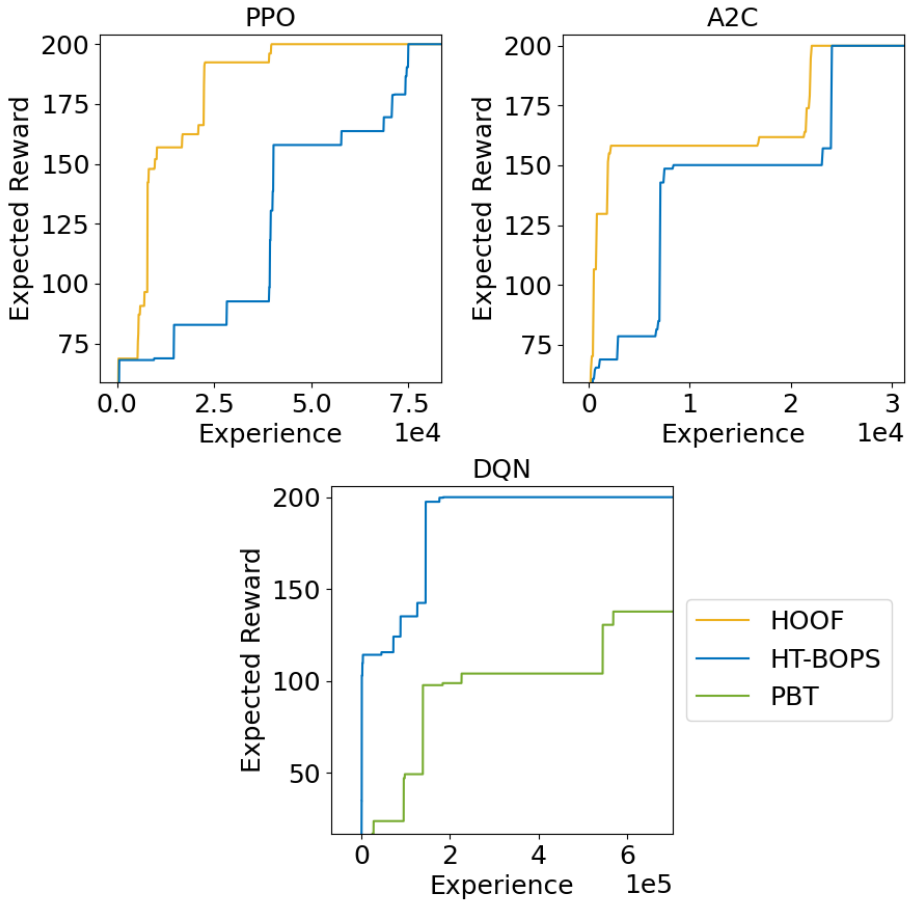

Figure 5.6: Performance of HT-BOPS compared to HOOF and PBT on CartPole

Learning Algorithm	Tuning Algorithm	Experience for N% of Maximum Reward 25%	50%	75%	90%
PPO	HOOF	100	7,900	8,200	22,400
	HT-BOPS	100	39,200	40,300	70,900
A2C	HOOF	200	500	1,900	21,900
	HT-BOPS	100	7,100	7,500	24,100
DQN	PBT	98,000	1.39e5	7.75e5	2.05e6
	HT-BOPS	900	900	1.45e5	1.45e5

Table 5.5: Experience to achieve various thresholds of maximum reward on CartPole

Across all thresholds, HOOF outperformed HT-BOPS by a wider margin for PPO than it did with A2C, while HT-BOPS significantly outperforms PBT for DQN. HOOF is likely to outperform HT-BOPS, in terms of convergence rate for all thresholds, when there is a high fraction of hyperparameter configurations that solve the environment. HT-BOPS requires the generation of an evaluation trajectory at each training iteration, while HOOF uses the trajectories generated for training to evaluate the hyperparameter configurations. This additional evaluation trajectory at each training iteration adds a significant amount of experience required to reach a threshold. The additional experience is not necessarily required for environments with numerous of high preforming hyperparameter configurations since the difference between these configurations is negligible. This extra experience therefore causes HT-BOPS causes delays in convergence rate on these high fraction environments. Thus, HOOF is the suggested tuning method for high fraction environments when applicable while HT-BOPS is a suitable substitute.

5.1.5 Hypothesis Five: Scarce High Performing Configurations

Within these experiments, two environments had a low fraction of high performing hyperparameter configurations. These low fraction environments are seen as more complex than the high fraction environments. The first environment to be examined is the discrete environment, LunarLander; the asymptotic reward results are shown in Figure 5.7.

HT-BOPS reaches a larger asymptotic reward than HOOF for PPO. However, HT-BOPS achieves a notably lower reward than HOOF for A2C. This performance will be further explained in Section 5.1.3. Additionally, PBT achieves a noticeably lower reward than HT-BOPS for DQN. None of the tuning strategies were able to reach the solved threshold defined by the environment within the experience limit for

DQN.

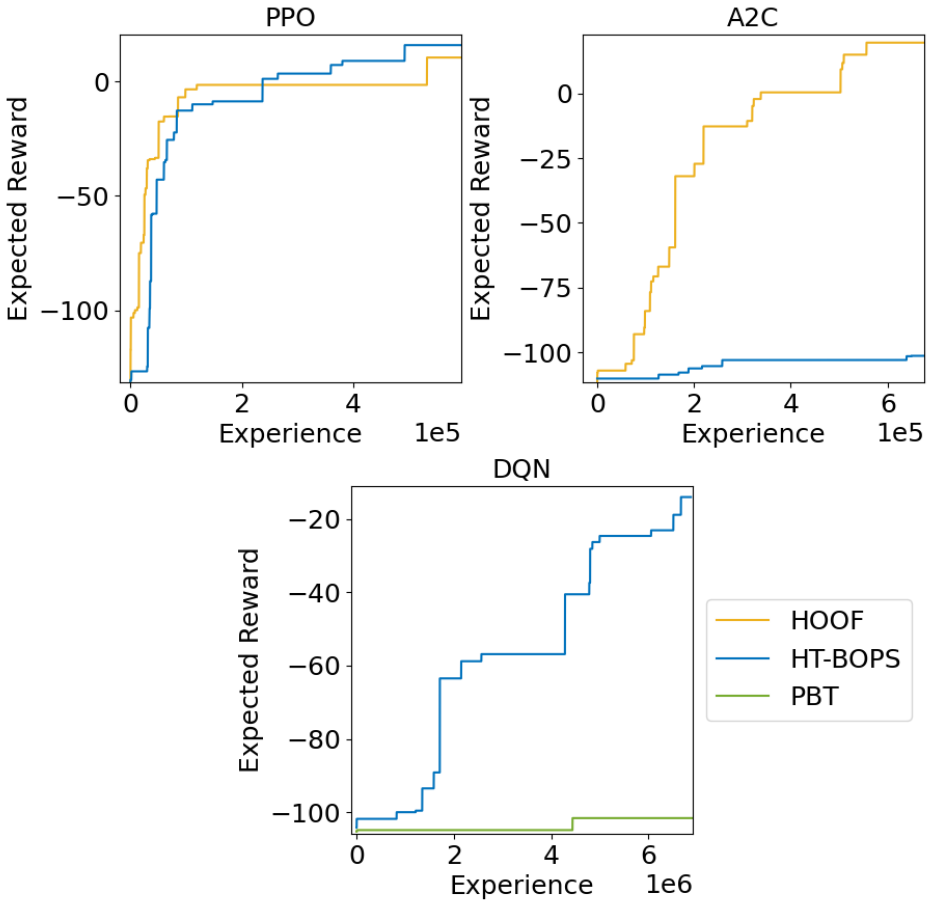

Figure 5.7: Performance of HT-BOPS compared to HOOF and PBT on LunarLander

Learning Algorithm	Tuning Algorithm	Experience for N% of Maximum Reward 25%	50%	75%	90%
PPO	HOOF	15,700	25,400	51,000	5.31e5
	HT-BOPS	35,500	37,500	83,400	3.60e5
A2C	HOOF	1.09e5	1.61e5	2.19e5	5.03e5
	HT-BOPS	N/A	N/A	N/A	N/A
DQN	PBT	1.51e8	1.59e8	1.77e8	N/A
	HT-BOPS	1.71e6	4.28e6	6.87e6	N/A

Table 5.6: Experience to achieve various thresholds of maximum reward on LunarLander

The summary of convergence rate at various thresholds can be found in Table 5.6. A faster convergence rate is implied when less experience is required to obtain that threshold of reward. When a configuration was not able to reach the reward threshold during the experience limit, a N/A was placed in the table instead of a value. In regards to PPO, HOOF has a faster convergence rate for the earlier phases of learning, $< 90\%$, while HT-BOPS then surpasses HOOF once the reward approaches the maximum. Since HOOF makes greedy selections, it can linger on local maximum for extended amounts of time. An example of this is shown here with HOOF for PPO. HOOF spends a significant amount of time with a maximum reward of 0, while HT-BOPS shows a consistently incremental increase as experience increases. HOOF found a local maximum quickly, which shows a faster convergence rate in early learning, compared to HT-BOPS due to its greedy nature. While HT-BOPS multi-policy and exploration method allows for consistent improvement as more experience is gathered which results in the faster convergence rate for late phases of learning.

The second low fraction environment is the continuous environment, Reacher. The asymptotic reward results for Reacher are shown in Figure 5.8. For every RL algorithm, HT-BOPS outperforms HOOF with respect to the asymptotic reward, since HT-BOPS is able to identify a larger maximum reward than HOOF. The convergence rate at various thresholds for Reacher is summarized in Table 5.7. Similar to that of

the LunarLander environment, HOOF has a higher convergence rate in early learning for all RL algorithms used in these experiments, while HT-BOPS again has a faster convergence rate to the larger rewards for PPO and A2C. Convergence to within 90% of the maximum reward for SAC was similar across HOOF and HT-BOPS.

Learning Algorithm	Tuning Algorithm	Experience for N% of Maximum Reward			
		25%	50%	75%	90%
PPO	HOOF	3.00e2	6.00e2	1.52e4	3.52e5
	HT-BOPS	3.00e2	2.30e3	1.77e4	6.31e4
A2C	HOOF	1.60e3	3.10e3	4.15e4	3.52e5
	HT-BOPS	7.00e2	8.80e3	5.05e4	8.13e4
SAC	HOOF	4.00e2	4.00e2	1.35e4	1.86e5
	HT-BOPS	4.2e3	2.12e4	1.45e5	2.03e5

Table 5.7: Experience to achieve various thresholds of maximum reward on Reacher

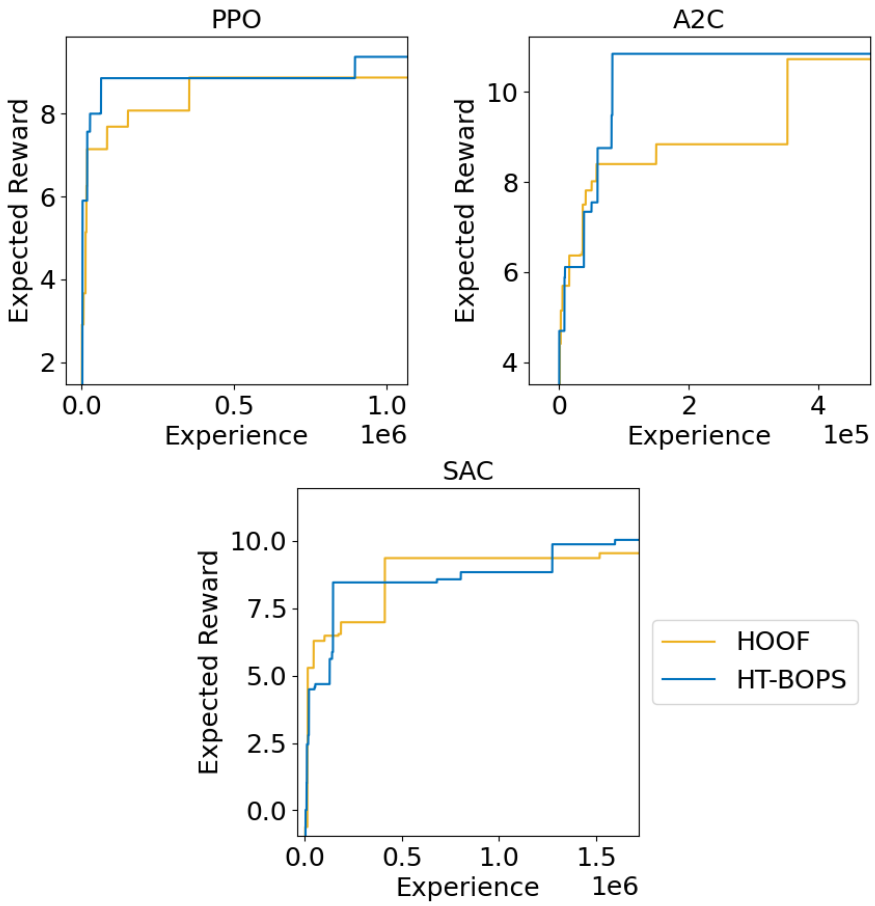

Figure 5.8: Performance of HT-BOPS compared to HOOF on Reacher

HT-BOPS identifies larger asymptotic rewards and converges to them faster than HOOF. Therefore, HT-BOPS is the suggested tuning method for low fraction environments. However, if experience is a constraint then HOOF may be the preferred tuning method since HOOF has a higher convergence rate for early learning.

5.1.6 Hypothesis Six: Hyperparameter Schedule vs Fixed Setting

The final hypothesis is focused on the performance difference of a fixed hyperparameter configuration and a schedule of hyperparameters. The hand tuned results utilize

a hyperparameter configuration that was found within documentation. These values are fixed for the entirety of training but may have been generated by a tuning procedure or through expert choice. HOOF and HT-BOPS both generate a hyperparameter schedule from a single pass of training data. A hyperparameter schedule is a choice of hyperparameter configuration at each iteration. First we examine the performance for the RL algorithm A2C whose results are shown in Figure 5.9.

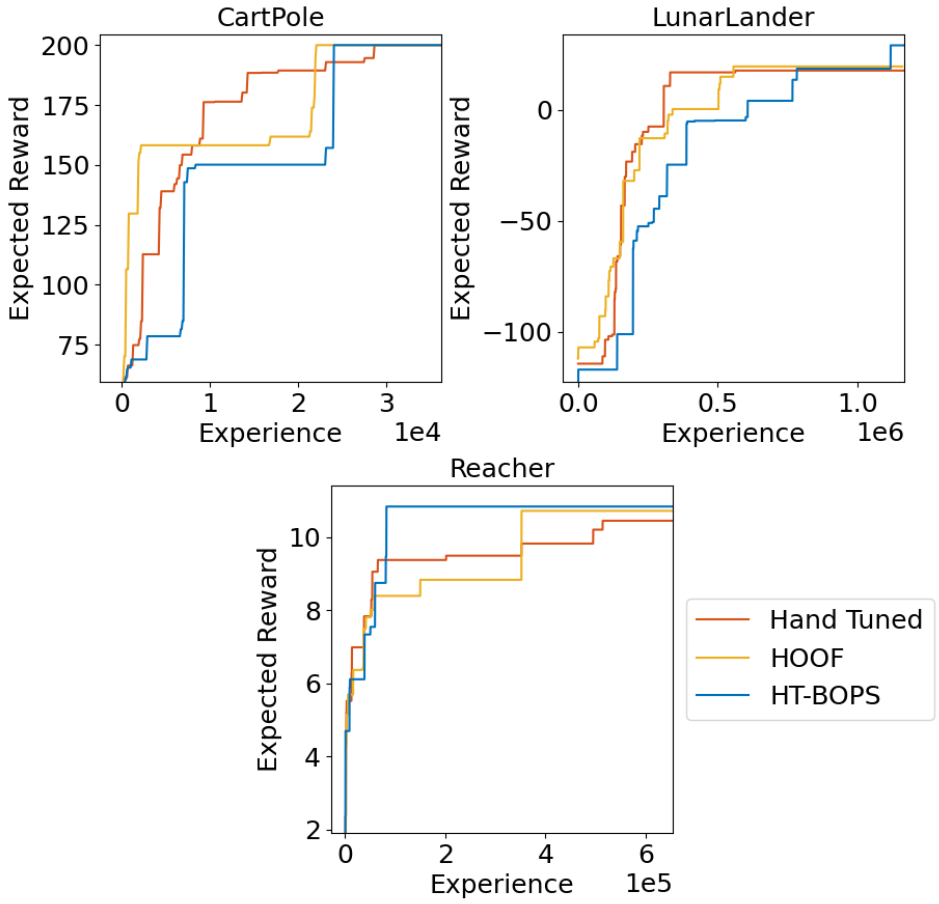

Figure 5.9: Comparison of Fixed and Schedule of Hyperparameter for A2C

The results for CartPole show that the hand tuned baseline has lower convergence rates in early phases of learning compared to HOOF while both HOOF and HT-BOPS are able to achieve the maximum reward with less experience on average. For

LunarLander, HOOF and the hand tuned baseline achieve similar performance while HT-BOPS is able to achieve a larger asymptotic reward on average. The results on Reacher show that convergence rate for early phases of learning are comparable across all three, while HOOF and HT-BOPS have higher convergence rates in later phases of learning as well as larger asymptotic rewards on average. Next, the results for PPO are shown in Figure 5.10.

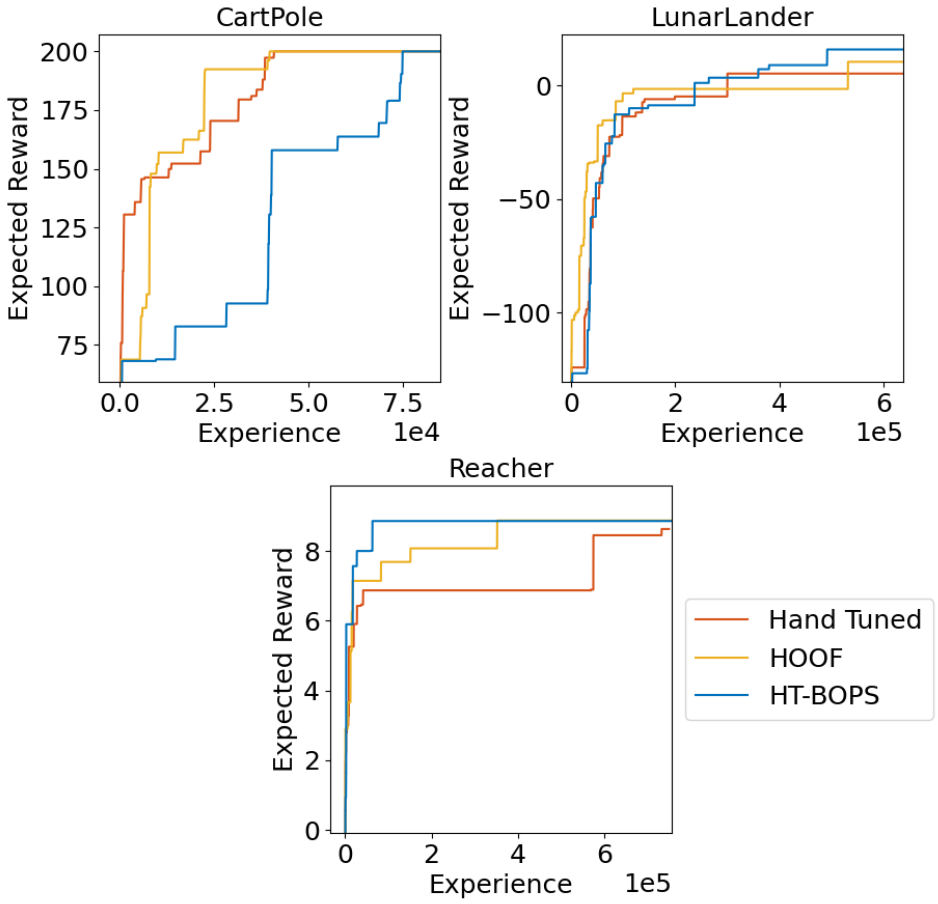

Figure 5.10: Comparison of Fixed and Schedule of Hyperparameter for PPO

For CartPole, HOOF and the hand tuned baseline have very similar performance while HT-BOPS convergence rates are lower for all phases of learning. All methods were able to achieve the maximum asymptotic reward. Again, for LunarLander the

performance of all three are similar in early phases of learning with HOOF being slightly better. HT-BOPS is then able to outperform in later phases of learning and HT-BOPS and HOOF are able to produce a larger asymptotic reward than the hand tuned baseline on average. Finally, for Reacher HT-BOPS and HOOF are both able to significantly outperform the hand tuned baseline. Finally, we have a valued based method, DQN, results shown in Figure 5.11. HOOF was not used in this case because it is not applicable to value based methods.

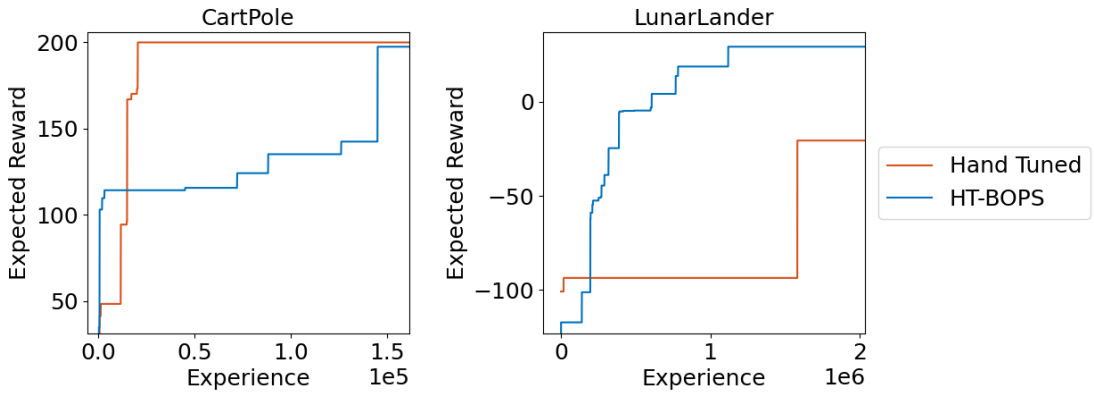

Figure 5.11: Comparison of Fixed and Schedule of Hyperparameter for DQN

For CartPole, the hand tuned baseline is able to significantly outperform HT-BOPS in terms of convergence rate while both are able to achieve the maximum asymptotic reward on average. While for LunarLander, HT-BOPS is able to significantly outperform the hand tuned basline for both convergence rate and asymptotic reward.

The tuned baseline was able to achieve comparable results to that of HOOF and HT-BOPS in some scenarios, however, HOOF and HT-BOPS are able to outperform the hand tuned baseline in the majority of scenarios. Therefore, a schedule of hyperparameter configurations can exceed or at least match the performance of a fixed hyperparameter configuration. Additionally, it is assumed that the hand tuned hyperparameter configuration was chosen based on some experience with the environment

or algorithm. However, since the exact strategy is unknown, the experience required to produce this hyperparameter cannot be accounted for.

5.2 Regret of Selected Hyperparameters

Regret is defined as the difference between the reward that is possible to achieve and the reward that was actually achieved [24]. In order to calculate true regret in the RL hyperparameter tuning setting, regret for every hyperparameter configuration for each N-step outlook would have to be calculated. This is necessary because the data generated at the next time step is dependent on the hyperparameter configuration choice at this time step, which could result in a better policy. However, this process is computationally expensive and thus a substitute is proposed. In this section, myopic regret, or short term regret, is used to estimate true regret.

Within this section, the "ideal" version of HOOF and HT-BOPS is defined by selecting the hyperparameter configuration that would result in the largest expected reward at the next evaluation. This concept is shown procedurally in Algorithm 15.

Algorithm 15 Single Step Myopic Regret

Input: Training iterations, T, learner set L, tuning method update procedure, P
Output: Regret, R

Initialize maximum reward record, M
Initialize selected reward record, S
for $i = 0, \ldots, T$ **do**
    Update policies through P
    Record tuning method choice of hyperparameter configuration
    Evaluate hyperparameter configuration through corresponding policy
    Identify maximum reward of evaluation, r_m
    Identify reward evaluation for selected hyperparameter configuration, r_s
    Record maximum reward seen, $M_i = max(M, r_m)$ and $S_i = max(S, r_s)$
end for
Calculate regret, $R = \frac{AUC(M) - AUC(S)}{AUC(M}$
return R

The learner set is a set of hyperparameters and their corresponding policies and

the tuning method update procedure is assumed to be P. At each training iteration, this procedure would update or generate a number of policies that correspond to the candidate hyperparameter configurations on line 4. The choice of hyperparameter configuration and corresponding policy is recorded on line 5. Then, all of the policies generated by the tuning procedure are evaluated on line 6. The evaluation consists of running evaluation trajectories and recording the average cumulative reward. The maximum reward and the reward corresponding to the selected hyperparameter configuration from line 5 are identified on lines 7 and lines 8 respectively. The maximum reward for each record is identified and stored for the given training iteration on line 9. After all training iterations are complete, the myopic regret is calculated ,on line 11, by calculating the area under the curve, AUC, for both reward histories. The fractional regret is used for more comprehensible and comparable results.

This is considered myopic regret since the regret is based on a single step outlook. Two experiments were run for each tuning method in order to gain an understanding on how much regret each of these algorithm accumulate: A2C on CartPole and PPO on LunarLander. While myopic regret is not comparable across tuning methodologies, it still provides insight into how the tuning method is making decision and potential improvement.

5.2.1 HOOF

HOOF is a greedy tuning methodology since it greedily selects the best reward estimate at each decision point. The myopic regret curves for HOOF are shown for CartPole and LunarLander in Figure 5.12.

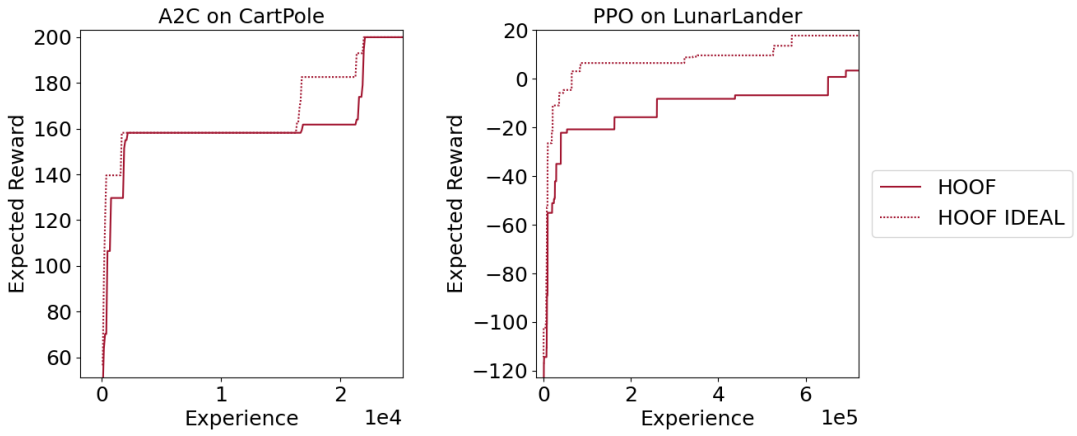

Figure 5.12: Myopic Regret Curves for HOOF

In order to calculate the regret estimate, the area between the two curves was calculated and is represented as a percentage of the ideal as shown in Table 5.8. Additionally, the percentage of choices that matched the ideal selection was tabulated. While HOOF on CartPole had very little myopic regret, 1.07%, the percent of ideal choices made was significantly lower than expected at 8.82%. Since CartPole has a large fraction of high performing hyperparameter configurations, it is likely that multiple configurations may result in the same or similar rewards. This would result in minimal difference in behavior but a reduced fraction of ideal decisions being made.

Environment	Learning Algorithm	Myopic Regret Percentage	Percent of Ideal Choices Made
CartPole	A2C	1.07%	8.82%
LunarLander	PPO	14.40%	6.41%

Table 5.8: Cumulative Myopic Regret for HOOF

HOOF performance on LunarLander generated more myopic regret and an even lower fraction of ideal choices being made. While LunarLander has a low fraction of high performing hyperparameter configurations, the likely cause of myopic regret for HOOF on LunarLander is the reuse of training experience to produce the policy

estimate. Training experience produces a weaker signal of the policy's performance if evaluation experience was used to generate the estimate. Although HOOF had a low percentage of choices that were equivalent to the ideal choices, having a myopic regret percentage less than 15% for both experiments shows good performance and balance of exploration and exploitation.

5.2.2 HT-BOPS

HT-BOPS is a methodology that focuses on exploration while managing multiple simultaneous policies. The myopic regret curve for HT-BOPS is shown for A2C on CartPole and LunarLander in Figure 5.13. As noted in Table 5.9, HT-BOPS on CartPole with A2C has a myopic regret percentage of 3.08% while making the ideal choice 15.25% of the time. Since the maximum reward is shown, it is likely that HT-BOPS made a choice with high regret at single points while making ideal choices from then on since the percentage of ideal choices made is higher than previously seen in these experiments. Seeing that HT-BOPS utilizes a windowed UCB solution, the inclusion of variance and use count within the window may force a sub-optimal configuration choice. This sub-optimal choice may be the cause for some of these in higher regret steps.

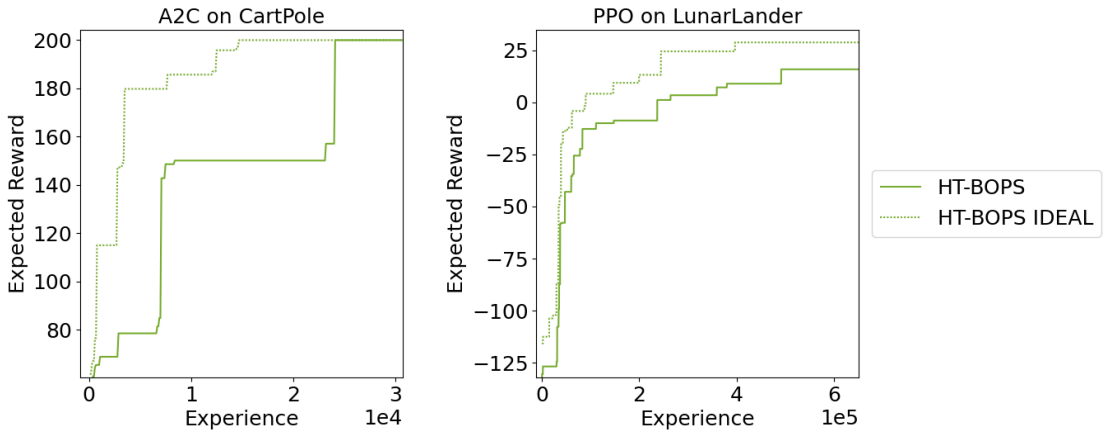

Figure 5.13: Myopic Regret Curves for HT-BOPS

In early phases of learning for LunarLander, HT-BOPS is follows closely to the ideal curve without being explicitly greedy. HT-BOPS had 7.12% of choices that were equivalent to the ideal choices, while having a myopic regret percentage less than 12%. The results for both experiments shows good performance and balance of exploration and exploitation.

Environment	Learning Algorithm	Myopic Regret Percentage	Percent of Ideal Choices Made
CartPole	A2C	3.08%	15.25%
LunarLander	PPO	12.04%	7.12%

Table 5.9: Myopic Regret for HT-BOPS

While myopic regret is not comparable across tuning methodologies, the percentage of ideal choices made is. In this aspect, HT-BOPS was able to outperform HOOF for both experiments. As previously stated, this is likely due to the difference in data that is used to generate the performance estimates of the hyperparameter configurations. HOOF reuses the training samples while HT-BOPS utilizes a history of evaluation trajectories while generating a new trajectory at each decision point.

5.3 Summary

It is unlikely that a single tuning method is preferred for every environment and every RL algorithm. In this chapter, I have shown that HT-BOPS outperforms HOOF on average, while also identifying situations where HOOF should be the preferred tuning method. To more clearly describe when each tuning method should be used, a decision tree is shown in Figure 5.14. PBT was left out of the decision tree due to its large experience requirements, which made it significantly under perform compared to HT-BOPS and HOOF. On average, RL algorithms with clipped gradients or proximal updates perform well on many common RL baselines relative to those that do not have such restrictions. Therefore, we recommend using such RL algorithms with HT-BOPS as a good starting point for most cases.

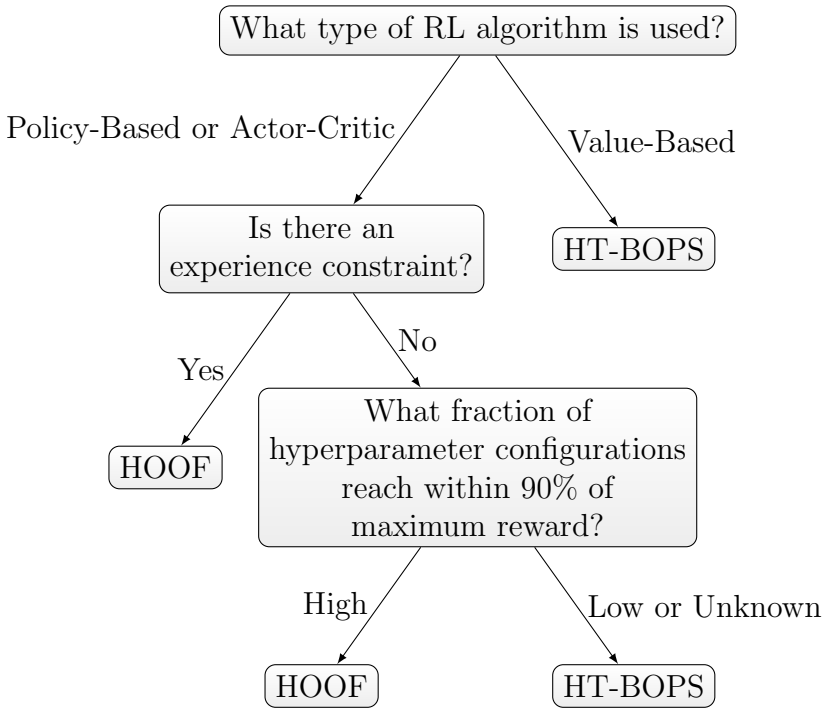

Figure 5.14: Tuning Method Selection Decision Tree

Chapter 6

Conclusion and Future Work

This work first demonstrates that hyperparameter tuning is a necessary component of any Reinforcement Learning (RL) experiment because the performance of a RL algorithm is highly sensitive to the configuration of its hyperparameters. However, it is critical that the tuning procedure is also efficient in terms of time and samples and thus a method that can tune in a single pass is highly desirable. The survey of the treatment of hyperparameter tuning for RL algorithms provided background on some state of the art tuning methodologies, however, a gap was identified. In order to fill this gap and provide a competitive single pass tuning method, this thesis frames the hyperparameter tuning in RL as a non-stationary multi-armed bandit problem. The presented tuning method, HT-BOPS, utilizes a windowed upper confidence bound algorithm to select the arm and a particle filter to estimate policies using off-policy data. HT-BOPS is sample efficient relative to start of the art methods, like PBT, since it reuses samples collected by one hyperparameter configuration to estimate the value of another.

My results show that HT-BOPS often produces the best rewards and competitive convergence rates across multiple algorithms and domains compared to other state-of-the-art baselines. However, I have shown that no single existing tuning method is

best for all scenarios. There exists a preferred tuning method for various categories of RL learning algorithms and environments.

While HT-BOPS has shown significant promise as a tuning method, future work remains. First, HT-BOPS has its own three parameters, or hyper-hyperparameters, as described in Section 4.2.4. Fixed values were used in all of the experiments presented in this thesis, however, a sensitivity analysis is required. Two of the three HT-BOPS parameters are: windows size and the number iterations before policy exploitation. A significant improvement to HT-BOPS would be if these hyper-hyperparameters were dynamically chosen. The dynamic choice of these parameters would result in HT-BOPS only have a single parameter, the bounding probability for the windowed UCB, which could be intuitively chosen. Additionally, theoretical bounds could be developed to frame the convergence rates during various types of learning. Next, HT-BOPS could include a KL constraint external to the RL algorithm in order to handle the variance in IW associated with diverging properties. Another improvement to HT-BOPS would be the inclusion of the modified update rules for offline RL. The update rule would be used for all hyperparameter configurations and corresponding policies that were not used to generate the training data for that iteration. This would allow for proper policy updates in order to restrict divergent policies and misbehavior such as oscillating polices. Finally, in addition to hyperparameter tuning for RL algorithms, the key ideas of HT-BOPS could be adopted to a variety of other settings.

The guide to tuning strategies proposed in this work should be used for any RL experiment to assure maximal performance with respect to the asymptotic reward or convergence rate.

Bibliography

[1] Shipra Agrawal and Navin Goyal. Analysis of Thompson Sampling for the Multi-Armed Bandit Problem. In Shie Mannor, Nathan Srebro, and Robert C. Williamson, editors, *Proceedings of the 25th Annual Conference on Learning Theory*, volume 23 of *Proceedings of Machine Learning Research*, pages 39.1–39.26, Edinburgh, Scotland, 25–27 Jun 2012. JMLR Workshop and Conference Proceedings.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47:235–256, 05 2002.

[3] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems*, pages 81—-93. IEEE Press, 1990.

[4] James Bergstra and Yoshua Bengio. Random Search for Hyperparameter Optimization. In *Journal of Machine Learning Research*, volume 13, pages 281–305. University of Montreal, Jan 2012.

[5] Vincent François-Lavet, Raphael Fonteneau, and Damien Ernst. How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies. In *NIPS 2015 Workshop on Deep Reinforcement Learning*, 2015.

[6] Aurélien Garivier and Eric Moulines. *On Upper-Confidence Bound Policies for Non-Stationary Bandit Problems*, pages 174–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870, Stockholmsmässan, Stockholm Sweden, July 2018. PMLR.

[8] D.G. Horvitz and D.J. Thompson. A Generalization of Sampling Without Replacement from a Finite Universe. In *Journal of the American Statistical Association*, volume 47, pages 663–685. Taylor & Francis, 1952.

[9] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks, 2017.

[10] Nan Jiang and Lihong Li. Doubly Robust Off-policy Value Evaluation for Reinforcement Learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 652–661, New York, New York, USA, June 2016. PMLR.

[11] Michail G. Lagoudakis and Ronald Parr. Model-free least-squares policy iteration. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14, pages 1547–1554. MIT Press, 2002.

[12] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous Control with Deep Reinforcement Learning, 2019.

[13] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1928–1937. University of Montreal, 2016.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. In *NIPS 2013 Workshop on Deep Learning*, 2013.

[15] Pierre Del Moral. Nonlinear filtering: Interacting Particle Resolution. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 325(6):653–658, 1997.

[16] Tom Le Paine, Cosmin Paduraru, Andrea Michi, Caglar Gulcehre, Konrad Zolna, Alexander Novikov, Ziyu Wang, and Nando de Freitas. *Hyperparameter Selection for Offline Reinforcement Learning.* Now Foundations and Trends, 2012.

[17] Jack Parker-Holder, Vu Nguyen, and Stephen Roberts. Provably Efficient Online Hyperparameter Optimization with Population-Based Bandits. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 33. Curran Associates, Inc., 2020.

[18] Supratik Paul, Vitaly Kurin, and Shimon Whiteson. Fast Efficient Hyperparameter Tuning for Policy Gradient Methods. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in*

Neural Information Processing Systems, volume 32, pages 4616–4626. Curran Associates, Inc., 2019.

[19] M.J D. Powell and J. Swann. Weighted Uniform Sampling — a Monte Carlo Technique for Reducing Variance. In *IMA Journal of Applied Mathematics*, volume 2, pages 228–236, 09 1966.

[20] D.B Rubin. Using the SIR Algorithm to Simulate Posterior Distributions. In D. V. Lindley J. M. Bernardo, M. H. DeGroot and A. F. M. Smith, editors, *Bayesian Statistics 3*. Oxford University Press, 1988.

[21] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, July 2015. PMLR.

[22] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[23] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017.

[24] Shai Shalev-Shwartz. Online Learning and Online Convex Optimization. In *Foundations and Trends in Machine Learning*, volume 4, pages 107–194, 2012.

[25] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25, pages 2951–2959. Curran Associates, Inc., 2012.

[26] Masashi Sugiyama, Hirotaka Hachiya, and Tetsuro Morimura. *Statistical Reinforcement Learning: Modern Machine Learning Approaches*. Chapman &Hall/CRC, 1 edition, 2013.

[27] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[28] Philip Thomas and Emma Brunskill. Data-Efficient Off-Policy Policy Evaluation for Reinforcement Learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2139–2148, New York, New York, USA, June 2016. PMLR.

[29] Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-Gradient Reinforcement Learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 2396–2407. Curran Associates, Inc., 2018.

[30] Tong Yu and Hong Zhu. Hyperparameter Optimization: A Review of Algorithms and Applications, 2020.