Faculty of Science, Technology and Communication

Reinforcement Learning for Online Hyperparameter Tuning of Complex Systems

Thesis Submitted in Partial Fulfilment of the Requirements for the Master's Degree in Information and Computer Sciences

Author:
Yann Hoffmann

Supervisor:
Prof. Martin Theobald

Reviewer:
Prof. Christoph Schommer

Advisor:
Benedikt Wilbertz

Second Advisor:
Mounir Boudia

July 2021

Abstract

Hyperparameter tuning of complex systems is a problem that relies on many decision-making algorithms and hand-crafted heuristics. Developers are often called upon to tune these hyperparameters by hand using their best judgment. But over time, systems experience a natural drift and configurations become obsolete. The maintenance of such systems is expensive, does not scale well, and calls for what is known as autonomic computing — the ability of networks to tune themselves without outside intervention.

In this thesis, we explore the use of reinforcement learning (RL) as an autonomic computing solution for complex systems. We use reinforcement learning as a zeroth-order optimizer for systems abstracted as black boxes. Implementing RL agents is a two-step process that comprises tuning the underlying model and training the agent on the task. When properly executed, the RL agents are capable of successfully enhancing a wide range of heuristics and even creating new ones from scratch. Moreover, we show that decision tree approximators can be used on top of Q-learning to handle large state spaces or improve the ability of agents to generalize from unknown samples. Several adjustments were made to usual RL training algorithms to accommodate for decision tree ensembles in an online learning setting. The latter part of the thesis sets out to build a faithful simulation for streaming processing, an instance of a complex system. The simulation follows sound concurrency principles and decouples the environment from the agent by using a message-passing library. Once in the simulation, the pipelines are optimized by a series of agents subject this time to a multi-objective reward function. Throughput and latency are chosen as the competing metrics to optimize for. Using this formulation, we introduce the multi-dimensional equivalent of Q-learning, namely $\mathring{Q}$-learning. After which we study the behavior of agents in the dynamic environment and rank them according to their performance. Agents are shown to converge quickly, adapt to the environment drift, and are robust to outside disturbances.

Acknowledgments

I would like to thank the team at the company for trusting me on this challenging yet rewarding project during my 6-month long internship. The thesis topic allowed me to immerse myself in a large and fascinating subfield of machine learning. I have come out of the project with a distinctively different view on engineering projects and am bubbling with ideas. Along the way, I have been exposed to many new concepts that have encouraged me to re-evaluate my work, and for that I am grateful.

I am especially indebted to Mounir, my advisor, for his unwavering confidence in the project and the many invaluable conversations we had. It would be an understatement to say that the project would not have been possible without him. He was there to nudge me in the right direction, to give me the colloquial new perspective on an old problem, and to point out a solution where none was to be expected. Thank you for the support.

Finally, I would like to express my gratitude to all those that have contributed to this thesis and made it possible. This thesis concludes an exciting chapter of my life and I am looking forward to an equally exciting one.

Contents

1 Introduction **vi**

1.1 Motivating the Use of Reinforcement Learning as an Optimization Procedure vii

1.2 Challenges of Reinforcement Learning and Remedies viii

1.3 Reinforcement Learning Primer ix

1.3.1 Markov Decision Processes ix

1.3.2 Generalized Policy Iteration Framework x

1.3.3 Generalized Returns: From TD to Monte Carlo Methods xi

1.3.4 Q-learning and Value Backup xii

2 Literature Review **xiv**

2.1 Related Works xiv

2.2 SmartChoices xvii

2.2.1 Algorithms Used xvii

2.2.2 Applications xvii

3 Synthetic validation **xix**

3.1 Binary Search xix

3.1.1 General problem statement xix

3.1.2 Description of the Environment xx

3.1.3 Analytic Baseline xxi

3.1.4 Agent and Environment xxii

3.2 Experimental Results xxiv

3.2.1 Results xxiv

3.2.2 Model Interpretation . . . xxvi
3.3 Conclusion . . . xxviii

4 Simulation **xxix**

4.1 Methodology and Benchmarking . . . xxix
4.1.1 Motivation for Building a Simulation . . . xxix
4.1.2 Processing Benchmarking . . . xxx
4.2 Solution Architecture . . . xxx
4.2.1 Optimizer API . . . xxx
4.2.2 Pipelines, Processing Times, and Drifts . . . xxxi
4.2.3 Message Passing . . . xxxii
4.2.4 Concurrency Considerations . . . xxxii
4.3 Reinforcement Learning Environment . . . xxxiv
4.3.1 Local Agents, Global Agents, and Omniscient Agents . . . xxxiv
4.3.2 Exploration Model . . . xxxiv
4.4 Optimization Objective: Latency and Throughput . . . xxxv
4.4.1 Definitions . . . xxxv
4.4.2 Intuition Behind the Optimal Learned Behavior . . . xxxvi
4.5 Convex Optimization . . . xxxvii
4.5.1 Problem Statement . . . xxxvii
4.5.2 Reward Shaping . . . xxxviii
4.5.3 Dual Optimization . . . xxxviii
4.5.4 Q-learning with Convex Hull . . . xxxix
4.6 Experimental Results . . . xlii
4.6.1 Qualitative Results . . . xliii
4.6.2 Quantitative Results . . . xlv
4.7 Further Improvements . . . xlv
4.8 Conclusion . . . xlvii

Bibliography **xlix**

References . . . xlix

Appendices **li**

A Additional material **lii**

Chapter 1

Introduction

Keywords— Reinforcement Learning, Autonomic Computing, Q-Learning, Multi-Objective Optimization, Gradient Boosted Machines

The status quo of infrastructure management is that a majority of configurations and parameters are tuned by hand. This manual approach works well in many circumstances but suffers from several unescapable problems which have prompted the writing of this thesis. The truth is that most, if not all, parameters that are set by hand yield a suboptimal performance. Finding the perfect configuration requires a lot of resources, specialized knowledge, and a good awareness of the system. Even if an efficient configuration can be found, it is never long before the system changes again and the configuration becomes obsolete. Moreover, systems scale but hand-tuning does not. Ad hoc tuning can quickly become a bottleneck and a source of unwanted expenses in a growing system.

On the other hand, autonomic computing holds out the promise of optimally operating any system, at a fraction of the price, and to scale horizontally as well as vertically. The term autonomic comes from the Greek root *auto- nomous* meaning "having its own laws". Autonomic is used, for instance, to describe the involuntary biological responses of the human body. It gives the impression of a controller, discreet and powerful, that can efficiently operate changes in a complex system all by itself. This vision of autonomic computing, first articulated by Jeff Kephart, would solve many of the current problems of infrastructure management.

The primary objective of my thesis is to apply reinforcement learning (RL) methods to optimize various hyperparameters in streaming pipelines. Pipelines can be thought of as a composition of functions where a certain number of parameters, assumed to be suboptimal, have to be injected at runtime. These can be static arguments, execution flags, or infrastructure-related variables. Regardless of the parameter, the RL agent sees the pipeline as a black box and endeavors to maximize the reward it gets from its environment as we will see in the next chapters.

1.1 Motivating the Use of Reinforcement Learning as an Optimization Procedure

A corollary of reasoning about black boxes is that the algorithm has no knowledge of the function being optimized. The algorithm can thus be expressed in very general terms. Optimization procedures used to optimize black boxes are known as derivative-free or zeroth-order optimization procedures. Reinforcement learning is only of many black-box optimizations that can be brought to bear on tuning pipelines. Random search, for instance, iteratively selects a random direction in the search space and moves to the optimum in that direction without knowing the function. Nelder-Mead is a popular refinement of the random search algorithm. It shifts a simplex in the search space according to a fixed set of rules until convergence. Although these are good example of black-box optimizers, meta-heuristics are fundamentally static algorithms that cannot tune online systems.

Bayesian optimization (BO) goes a step beyond random search procedures and incorporates Gaussian processes as its local model of the objective function. Bayesian optimization is a competitive option and has been successfully applied by Twitter to tuning the JVM. However, BO does not satisfy all the requirements because determining the posterior in BO is particularly computationally intensive.

What sets reinforcement learning apart from the other methods is its suitability when applied to dynamical systems. The wide range of variations in Q-learning algorithms and the fact that Q-learning can be coupled with various function approximators are decisive factors when choosing the optimizer for complex systems. Throughout the literature, deep reinforcement learning is referred to as the state of the art of complex control tasks.

1.2 Challenges of Reinforcement Learning and Remedies

Developing a reinforcement learning solution comes with a set of distinctive challenges. Dulac-Arnold et al. have identified 9 of these as they apply to productionizing a reinforcement learning solution (Dulac-Arnold, Mankowitz, & Hester, 2019). We have decided to address them as early as possible because they have played such an important role in motivating the design decisions of the project.

1. *Training off-line from fixed logs of an external behavior policy.* N/A

2. *Learning on the real system from limited samples.* This challenge hides two problems. One relates to the cost of obtaining exploratory samples in a real-world system. The other to the sample inefficiency of RL techniques. As a whole, our approach has focused on building a simulation that can be used to freely collect samples. This is part of the idea of modeling an environment to improve the sample efficiency by planning. Once the agent is well-trained in a simulation, it can be safely graduated to a production environment.

3. *High-dimensional continuous state and action spaces.* A blend of function approximation, feature engineering, and special-purpose models is generally used to address these problems. In our case, we have purposefully restricted our analysis to discrete bounded space as the search space. This limits the severity of challenge 3.

4. *Safety constraints that should never or at least rarely be violated.* Safety constraints are hard to anticipate in the context of black-box optimization. Guardrails are sometimes installed for agents in the literature. We have decided to devolve this task to the user who is asked to explicitly define bounds on the search space.

5. *Tasks that may be partitially observable, alternatively viewed as non-stationary or stochastic.* The non-stationarity of the environment is one of the fundamental assumptions of this thesis. We have tried to address this challenge by using algorithms with a learning rate adapted to dynamic environments. The use of a jumping process has complemented the approach by acting as a regularizer.

6. *Reward functions that are unspecified, multi-objective, or risk-sensitive.* Multi-objective objec-

tive functions arise naturally in the treatment of pipelines in the form of latency and throughput. We have decided to let the user input the preference in relation to both rewards. The agent is then tasked to implement this preference in the simulation.

7. *System operators who desire explainable policies and actions.* At a basic level, the agents rely on decision trees, which are some of the more explainable policies. Interpretability of clusters of agents is however beyond the scope of this thesis.

8. *Inference that must happen in real-time at the control frequency of the system.* This point has motivated early on our choice of XGBoost as the function approximator. XGBoost is known for its training and inference time speed.

9. *Large and/or unknown delays in the system actuators.* We experimented with n-step rewards, and different discount rates to account for long delays in the system's response.

The paper goes on to give suggestions as to how to tackle these problems. In the main, our system passes most of the checklist, while some points are left for further work. Problems (2), (6), and (8) were given more attention than problems (9) or (5) because the latter two are substantially more complex issues.

The following paragraphs are dedicated to introducing useful RL concepts for the rest of the thesis.

1.3 Reinforcement Learning Primer

1.3.1 Markov Decision Processes

Reinforcement learning is a problem of sequential decision-making where an agent interacts with an environment and receives feedback in the form of a reward. Through explorative trial and error, the agent builds a policy π — the mapping from states to actions — so as to maximize its expected rewards in its environment. Finding the optimal policy denoted as $\pi^\star$ is the ultimate objective of control theory and reinforcement learning.

$$\pi : \mathcal{S} \to \mathcal{A} \tag{1.1}$$

A pathway to finding an optimal policy π^* is to learn the value of all the states $V_\pi(s)$ and select the states that have the highest value. By definition $V_\pi(s)$ is the expected sum of discounted rewards

$\mathbb{E}_\pi[G_t]$ where $G_t = \sum_{i=0}^{\infty} \gamma^t r_t$ is the expected return of following π in state s. Unlike supervised learning, there are no a priori labels of the value of the state which could be used to train a policy. The only guide is the reward signal. As we have seen, the value of a state $V_\pi(s)$ depends on the policy followed by the agent. This dependence makes it impossible to apply supervision because the policy is chosen based on the state values and vice versa. The interplay between value function and policy is further explored in Section 1.3.2 about General Policy Iteration.

All the previous considerations can be succinctly captured by a Markov Decision Process (MDP) $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma)$ containing respectively the states, actions, transitions as described by $p(s', a' \mid s, a)$, and discount factor γ. An MDP exhibits the Markov property because the latest state encapsulate all available information.

1.3.2 Generalized Policy Iteration Framework

The generalized policy iteration (GPI) framework describes how all RL algorithms that rely on a policy and a value function interact.

Moving the policy to its optimum is a two-step process. First, the **policy evaluation** makes the value function consistent with the current policy. Recall that because the value of a state is the expected return from being in that state, the value must be updated by the policy evaluation every time the policy is modified, hence the name. Conversely, the **policy improvement** makes the policy greedy with respect to the state values. If $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ then $v_{\pi'}(s) \geq v_\pi(s)$ and π' is an improvement on π through what is called the policy improvement theorem (section 4.2 RL). π' is generally chosen to be greedy with respect to the state values.

A policy that is greedy for all its state is by definition optimal because it cannot be improved. The policy is said to converge to its optimum once the policy π stays the same after one round of policy improvement. Indeed, the policy stays identical iff $\forall s, v_{\pi'}(s) = v_\pi(s)$, which means that the policy $\pi = \pi'$ cannot be made any greedier. The whole process therefore stabilizes at $\pi = \pi^*$ and $V = V^*$.

The generalized policy improvement features in many reinforcement learning algorithms including Q-learning. Briefly understanding how it applies to Q-learning gives us a hint as to how Q-learning is able to solve the reinforcement learning problem. In its implementation, the agent acts in state s

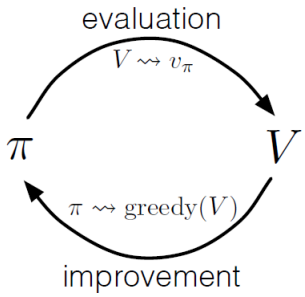

Figure 1.1: Illustration of the generalized policy improvement [Sutton and Barto, 1998]

by selecting action $\arg\max_a Q(s,a)$. We can immediately see that the policy is made to be greedy [1] with respect to the value function Q. This corresponds to the policy improvement.

$$Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') \tag{1.2}$$

Equation 1.2 describes the Q-learning update called at the end of an episode. The TD(0) target on the right-hand side can be rewritten as $r + \gamma V_\pi(s')$. The discounted part of the TD target matches the Q-learning policy, so we can see that the values are indeed made to be consistent with respect to the greedy policy. This corresponds to the policy evaluation. Q-learning is an example of an algorithm where the policy improvement is every time a transition is sampled.

1.3.3 Generalized Returns: From TD to Monte Carlo Methods

Monte Carlo (MC) usually refers to approximate inference techniques based on numerical sampling. In the context of RL, Monte Carlo methods designate algorithms that estimate state values by averaging sampled returns and that use the returns from the full episode to estimate the state value instead of 1-step transitions.

Monte Carlo can update the state value by considering only the first time a state is being visited in an episode, this is called first-visit Monte Carlo. The update can also be made regardless if the agent visits a state multiple times, this is called every-visit Monte Carlo. The every-visit variation is the one used by Q-learning.

Finally, Monte Carlo are fundamentally offline methods because of the episodic nature of the

[1] In reality and without loss of generality, the policy of the agent is epsilon-greedy with respect to the values. As explained in section 1.3.4, Q-learning is an off-policy algorithm, which means that the policy of the policy evaluation differs from the policy of the policy improvement.

returns. To know the exact return of a state we have to wait until the end of an episode. Yet we wish to tune hyperparameters in an online fashion with infinite-time horizons. This can be solved by introducing the truncated or partial return $G_{t:t+n} = R_t + \gamma R_{t+1} + \cdots + \gamma^{n-1} R_{t+n-1} + V(s_{t+n})$. The last value of the return $V(s_{t+n})$ is relied on to give of an approximation of the geometric tail $\sum_{k=n}^{\infty} \gamma^k R_k$ that would normally be there in a Monte Carlo method. Using an approximate value of $V(s_{t+n})$ to update $V(s)$ is called bootstrapping. By using the general form of an n-step return, we can have a more flexible Q-learning algorithm that is better adapted to non-stationary environments.

In summary, Q-learning uses a one-step look-ahead $G_{t:t+1} = R_t + \gamma V(S_{t+1})$ while Monte Carlo methods use the full return with no need to bootstrap. This is equivalent to $G_{t:\infty} = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$ where all rewards after the terminal state are set to 0.

1.3.4 Q-learning and Value Backup

We are now ready to build our first approximate dynamic programming algorithm. Q-learning regroups most of the ideas we have previously encountered (Watkins & Dayan, 1992):

Algorithm 1 Q-learning algorithm

```
Initialize π
for episode = 1, M do
    Initialize s
    for step = 1, T do
        Choose a following the ϵ-greedy policy π
        Observe r, s'
        // Backup rule
        Q(s, a) ⇐^α r + γ max_{a'} Q(s', a')
        s ← s'
    end for
end for
```

At its core, the algorithm has a backup rule that specifies how to update the table estimates for action values. Here the backup rule uses a one-step look-ahead from transition (s, a, s', r): $Q(s,a) \overset{\alpha}{\Leftarrow} \delta$ where δ is called the target. The notation seen in the update $Q(s,a) \overset{\alpha}{\Leftarrow} x$ is used as a shorthand for $Q(s,a) \leftarrow x + \alpha(Q(s,a) - x)$. Because x is only an estimate of the true value of $Q(s,a)$, it is often better to use a mild learning rate α to update $Q(s,a)$ in the direction of x. It should be mentioned that Q-learning updates its values immediately after taking an action. A short GPI cycle makes the algorithm converge faster. The use of off-policy and bootstrapping can cause instability in the learning when used with function approximators. This is referred to as the deadly

triad and has to be dealt with on a case-by-case basis.

All value algorithms use the same blueprint to update their value estimates. What makes Q-learning unique is that it uses one-step TD updates on Q-values whereas SARSA for example uses a slightly different backup rule that is on-policy:

Q-learning target:

$$\delta_t = r_t + \max_{a'} Q(s', a')$$

SARSA target:

$$\delta = r + \gamma \begin{cases} \max\limits_{a'} Q(s', a') & \text{with probability } 1 - \epsilon \\ \text{random } Q(s', a') & \text{with probability } \epsilon \end{cases}$$

A reference to all the backup rules can be found in Figure A of the annex.

Chapter 2

Literature Review

2.1 Related Works

Research over the past two decades has produced a wide range of solutions to problems involving hyperparameter tuning. The research pertaining to our work can be divided into two groups: the tuning of infrastructure such as servers within data centers and the tuning of hyperparameters in a purely algorithmic setting. Although earlier works in the literature have gravitated towards solving both problems separately, the distinction between the infrastructure and the algorithmic is being blurred more recently by Carbune et al. Their work shows agents being applied to both everyday algorithms and infrastructure building (Carbune et al., 2019). We aim to replicate this general formulation throughout the next chapters.

A second trend in the literature is that of slowly moving away from heavily engineered models of the environment using queuing theory, reward shaping of the objective function, and feature engineering towards using deep reinforcement learning (DRL) models that obviate the need for these techniques. A lot of focus visible in earlier research is on initializing the agent with the proper Q-table or policy. These were meant to guide the agent in its early stages of learning. Learning from initialization is a basic form of imitation learning, a fertile research direction.

Some of the common themes include policy initialization occasionally guided by theoretical models of systems, parameter grouping, and online tuning. Most authors caution about the stability risks associated with DRL in a production environment.

By using a dedicated model based on queuing theory, Tesauro et al. manage to boost the performance of their Q-learning agent in order to optimize the resource allocation of a small data center.

At the core of their approach is the use of both DRL and systems modeling to compensate for what they name the poor scalability, the sample inefficiency, and the prohibitive cost of exploration of reinforcement learning (RL) in large state spaces (Tesauro, Jong, Das, & Bennani, 2006).

Unlike most reviewed papers, Li et al. commit to a particular platform. Each of the categorical actions available to the agent corresponds to an action in the AWS API like start a start, stop, and terminate. There is no need for an exploration model for hyperparameters because the authors only focus on the autoscaling of AWS resources through categorical actions. In addition, the authors make use of feature engineering by feeding moving averages of various metrics to the agent. Interestingly, they are the only ones that directly modify the Q-table of the agent instead of relying on an initialization policy. This technique is similar to optimistic initialization and the table is filled up according to a handful of rules (H. Li & Venugopal, 2011).

Some have been inspired by the work of Kephart et al. and define precisely the responsibility of the operator in autonomic computing. Quality-of-service concerns like latency and packet loss play a prominent role in what the operator defines as the service-level objective (SLO) of a platform. SLO introduces a shift in how the management of infrastructure. In particular, declaring high-level objectives can be compared to declarative forms of languages like SQL where the result, not the operations are described. The authors go on to define the reward function as the spread between the performance of the system and the SLO. They use a variety of techniques like initialization, policy refinement, and parameter grouping to train their agent to operate a cluster. In their methodology, they compare the DRL agent to several baselines, one random and one fixed at the default configuration of the VMs (Xu, Rao, & Bu, 2012).

Likewise, research has been made on tuning the configuration of Apache servers. Bu et al. use the default Apache configuration as a starting point and train a variety of agents in an off-policy manner. They inject some feature engineering to make the agent more performant in an overall very dynamic environment where new nodes, requests, and workloads are constantly being introduced. An interesting addition through their work is the use of guardrails for the agent. As soon as the agent crosses a particularly bad performance threshold, the policy of the agent is swapped to one of the default configurations (Bu, Rao, & Xu, 2009).

More recent research looks at tuning the configuration of MapReduce, Hadoop. Unlike most applications, the reward does not depend on the throughput or performance but on solely avoiding

excessive resource utilization. Frameworks like MapReduce come with their own benchmarking options like TeraSort and WordCount on which the agent is run (Peng & Zhang, 2017).

Whereas the work of Bu et al. is centered around virtual machines, the same interest for autonomic computing has now shifted towards micro-services. Coming from the private sector are a series of in-house reinforcement learning optimizers used to solve very general problems. In particular, Spiral (Facebook) was recently shown to be effective in everything from tuning micro-services, to automating caching heuristics, and spam filtering at scale via DRL. Amazon has also developed a closed-source solution for autoscaling gaming infrastructure based on predicted demand. Their work is very similar to Li et al.'s. Finally, Twitter is known to have obtained good results in tuning parameters of the Graal JVM. Unlike most, they have used Bayesian Optimization as the zeroth-order optimizer.

Xiang et al. have applied the particle swarm optimization (PSO) meta-heuristic in the same context (Liu, Li, & Duan, 2012). The solution suffers, however, from several downsides. PSO works well on static environments but requires constant reinitialization when working on live clusters. Whenever the workload changes the algorithm has to relearn the configuration, and the algorithm does not leverage data available in the environment.

Finally, *Learning to Optimize* stands out by teaching an RL agent to recreate a very specific type of heuristic, namely gradient-based optimization algorithms. The RL agent is shown traces of gradient $(\nabla_t, \nabla_{t-1}, \dots)$ and manages to reproduce popular algorithms like gradient descent, momentum, conjugate gradient, and L-BFGS. The agent occasionally deviates from these algorithms and outperforms them on a set of benchmark tasks. Much like SmartChoices, Learning to Optimize aims to enhance hand-engineered heuristics. In addition to the usual offline imitation learning, *Learning to Optimize* sets itself apart by leveraging guided policy search as its policy search method (Levine & Abbeel, 2014). Guided policy search is a compromise between model-free and model-based systems which ameliorates the sample inefficiency of model-free methods while avoiding the need for a specialized model of the environment (K. Li & Malik, 2016).

This first overview shows the variety of ideas and techniques used to perform hyperparameter tuning. In the next section, we turn our attention to SmartChoices whose formulation and applicability are the most general.

2.2 SmartChoices

SmartChoices is the second iteration of the paper *Predicted Variables in Programming* which aims to replace hand-crafted heuristics my data-driven RL agents (Carbune et al., 2018).The paper offers a good example of how to leverage deep reinforcement learning models to solve algorithmic problems or improve heuristic functions. For each one of three algorithms, the authors combine reward shaping with baselining to obtain performance gains measured on the relevant metric. They claim that their RL API is highly reusable and improves the maintenance of systems at scale. A SmartChoice substitutes an RL-enhanced heuristic for a handcrafted heuristic. In most cases, the RL-enhanced heuristic is not learned from scratch. Initializing the agent with a heuristic like the Least-Recently Used (LRU) policy often constitutes a good first approximation to problems like cache replacement. It is an effective bulwark against the many sensitivity issues of DRL, and heuristics can be combined to form a very safe and potentially more performant mixture of heuristics.

For all its advantages, SmartChoices does not directly address the problem of multi-objective optimization. The API gives the user the freedom to specify a reward vector and is said to find a Pareto solution. However, the user has no control as to which Pareto solution is chosen.

2.2.1 Algorithms Used

The models used by SmartChoices all rely on deep reinforcement learning. The first one, double deep Q-network (DDQN), is an improvement on deep Q-networks. Deep Q-networks are Q-learning agents fitted with a deep neural network as an approximation layer. DDQN agents are used in high dimension state spaces to predict discrete or categorical actions.

The second model, twin-delayed deep deterministic policy gradient (TD3), is a policy gradient variant that makes it suited for predicted continuous values.

2.2.2 Applications

The applications chosen by SmartChoices illustrate the fact that the reinforcement learning optimizer can be applied to both algorithmic problems and infrastructure problems in the pursuit of enhancing heuristics.

Binary Search

Binary search is the problem of finding an item in a sorted array. The canonical heuristic is the midpoint pivot selection which as the name indicates always selects the midpoint as the pivot. An alternative heuristic tailored to arrays generated from a uniform distribution is the interpolation. The interpolation heuristic selects the pivot based on the first and final index and the item searched. Both heuristics are used as a safety net for the RL agent. After pitting a mixture of heuristics against heuristic-less agents, we see that the agent without heuristic mix has the best long-term performance at the cost of a very high cumulative regret. The agents outperform nevertheless the heuristic baseline which demonstrates the effectiveness of data-driven solutions to improving the pivot selection heuristic.

QuickSort

QuickSort is a divide and conquer solution to sorting an array. Similarly to binary search, the agent has to choose a pivot along which to split the array. Choosing the median of the array as the pivot guarantees the quickest execution of the algorithm. In order to get closer to the median, the original heuristic can be extended by sampling without replacement n elements and taking the pivot as the sample median. The trade-off between computation cycles spent calculating the median of n elements and the advantage of having a pivot closer to the median is the focus of the SmartChoice agent. A non-trivial policy is obtained by training a DDQN agent.

Cache replacement

Lastly, cache replacement is shown to be a delicate problem. This is confirmed by the fact that 60% of the runs saw neither gain nor loss in performance from using DRL over the standard Least-recently used (LRU) strategy.

The various policy strategies are summarized in the following table.

Problem	Heuristic(s)	SmartChoice replacement
Binary Search	Midpoint selection, Interpolation	Mixture of heuristics
QuickSort	Random pivot, Median-of-n	Learn n in median-of-n
Cache replacement	Least Recently Used (LRU)	Direct prediction of evictions, Priority queueing

Chapter 3

Synthetic validation

Chapter 3 is the first step towards tackling autonomic computing applied to the problem of searching an item in an array. The objective of this chapter is twofold. First, we will reproduce some of the results obtained in SmartChoices and measure the performance of our agents. Second, we will validate the use of decision trees in Q-learning and analyze the tuning process of XGBoost.

3.1 Binary Search

3.1.1 General problem statement

This synthetic application is inspired in large measure by the first problem described in *SmartChoices*. The problem of binary search is one of finding an item inside a sorted array of size n with the least amount of lookups. Each lookup splits the array in two along the pivot and the search carries on the half-array which contains the item. Customarily, the pivot is chosen using a static heuristic of the midpoint index $p = \left\lfloor \frac{n}{2} \right\rfloor$, n being the size of the array. This consistent heuristic is simple to analyze and guarantees a complexity of $\mathcal{O}(\log(n))$. Can we do better than this by dynamically choosing a pivot with a reinforcement learning agent and to what extent does the performance of the model depend on the choice of function approximator?

The traversal dynamics of searching for item x in an array of size n can be captured by $f(a, b)$.

$$f(a,b) = \begin{cases} f(a, p-1) & \text{if } x < \text{array}[p] \\ f(p+1, b) & \text{if } x > \text{array}[p] \\ p & \text{if } x = \text{array}[p] \end{cases}, \text{ where pivot } p = \left\lfloor \frac{b-a+1}{2} \right\rfloor \tag{3.1}$$

3.1.2 Description of the Environment

In our environment, an episode starts off by generating a sorted array of size $n = 2^N - 1$. The items are sampled from one of several distributions with support in [0, 1]. Next, the environment randomly selects an item from the array. The agent has to find the index of that item in as few steps as possible.

During the episode, the agent is presented with information about the item being sought (eg. 5) and the subset of the array currently being searched in (eg. [0, 0.5], which represents the first half of the array). Finally, the agent can start enhancing the heuristic by selecting a pivot in [0, 1]. Either the pivot happens to be the item and the episode terminates with a reward of $+5$ or the agent receives a reward of -1 and gets ready for the next step. The agent is guaranteed to find x in at most $n - 1$ steps because each action shrinks the array by at least one.

To accomodate for decision trees, the action space [0, 1] has been discretized. There are a total of 11 actions to choose from $p \in \{0, 0.1, 0.2, \ldots, 1\}$.

Figure 3.1.3 illustrates the following trajectory that we get when applying the midpoint heuristic:

- $s_0 = [33, 0, 1], a_0 = 0.5, r_0 = -1$
- $s_1 = [33, 0, 0.5],\ a_1 = 0.5, r_1 = -1$
- $s_2 = [33, 0.25, 0.5], a_2 = 0.5, r_2 = -1$
- $s_3 = [33, 0.25, 0.375], a_3 = 0.5, r_3 = +5$

Note that the heuristic predictably chooses 0.5 as its action. After three steps, the item is found at position 4 terminating the episode with a reward of $+5$. As we will see, our agent gambles with the midpoint heuristic to try to find a quicker path to the item.

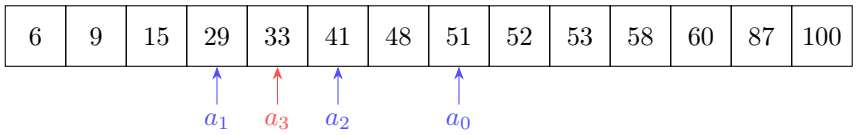

Figure 3.1: Sorted array.

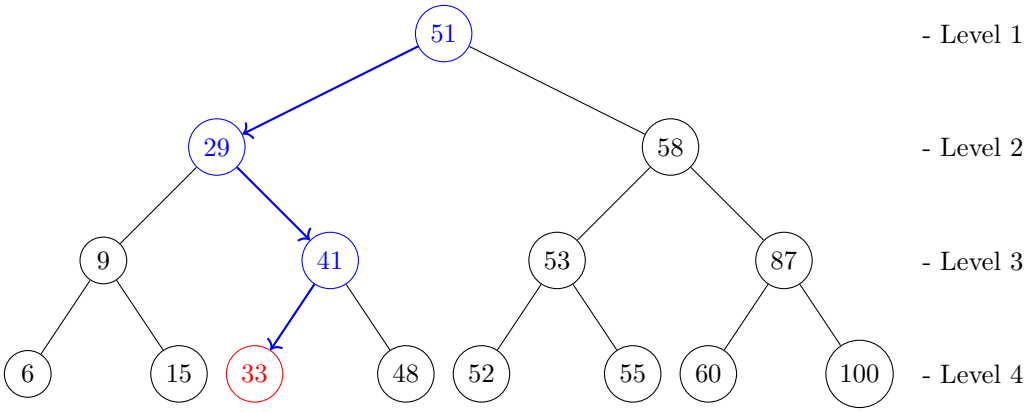

Figure 3.2: Balanced binary tree structure — optimal traversal when not knowing the distribution of the data.

3.1.3 Analytic Baseline

Let ξ be a random variable counting the number of iterations it takes for midpoint binary sort to find an item. The number of iterations ξ can also be seen as the level of the item in the tree of Figure 3.1.3. We can calculate a baseline for our agent by letting $N = 4$ be the height of the tree and n be the number of nodes in the tree: $n = 2^N - 1 = 15$. The expected length of an episode can be shown to be 3.27[1]. This result is used as a comparative baseline for the RL agents.

[1]

$$\begin{aligned}\mathbb{E}[\xi] &= \sum_{\text{level}=1}^{\text{max level}} \text{level} \cdot \frac{\#\text{level nodes}}{\#\text{tree nodes}} \\ &= \sum_{k=1}^{n} k \cdot \frac{2^{k-1}}{N} \\ &= 1 \cdot \frac{1}{15} + 2 \cdot \frac{2}{15} + 3 \cdot \frac{4}{15} + 4 \cdot \frac{8}{15} \\ &\simeq 3.27\end{aligned} \tag{3.2}$$

3.1.4 Agent and Environment

Gradient Boosted Machines

Decision trees provide a middle ground between exact tabular solutions and neural network-based approximations. On the one hand, tabular methods are a robust solution that is straightforward to train and interpret. Unfortunately, they quickly become inadequate as soon as the state space grows too large. On the other hand, decision tree-based function approximations have been shown to converge more reliably than neural networks on a battery of simple control problems like Mountain car and Pole balancing (Pyeatt & Howe, 1998). This suggests that decision trees could be better adapted overall to algorithmic problems like binary search. What is more, decision tree ensembles and XGBoost, in particular, have been widely used for large-scale applications. They are less prone to overfitting and less sensitive to outliers. (Sutton & Barto, 1992).

By following our approach, the burden has shifted from tuning the heuristic itself (ie. the choice of the pivot) to the tuning of parameters in the Q-function approximator. Far from being a circular methodology, the new set of parameters is arguably easier to tune, and as noted in SmartChoices, advances in AutoML make this step easier (Carbune et al., 2019). Our approach to tuning XGBoost was to sample configurations from the parameter space and rank the models according to their performance.

Incremental Training of Gradient Boosted Machines

Q-learning algorithms are trained on batches of data on which they regularly update their table or function approximator. Yet, decision tree algorithms (eg. CART, C4.5) do not support incremental training because all the data needs to be in memory before making the leaf splitting decisions. A workaround for this issue is described in the following procedure:

Algorithm 2 Incremental Q-learning for tree-based agents.

```
Initialize π
for epoch = 1, N do
    for episode = 1, M do
        Initialize s
        while s is not terminal do
            Choose a ∼ π(s)
            Add (s, a, s', r) to current batch b
            s ← s'
        end while
    end for
    π ← train(b)
    Empty b
end for
```

In this memoryless form, the procedure generates an alternating succession of policies and batches $(\pi_0, b_0, \pi_1, b_1, \pi_2, \ldots)$ where b_{i+1} depends on π_i and π_{i+1} on b_i. Building a new model on the previous batch of data circumvents the incremental training problem of decision trees. The only caveat with training π_{i+1} on the most recent batch is that previous batches may contain useful but are discarded.

The final iteration of the training algorithm addresses this problem. Like the previous algorithm, policies and batches are generated one after the other. The difference is that policies are now trained on all previous batches. Since the recent batches are more relevant to the policy, the batches are geometrically weighted by recency. If the common ratio of the weighting is set to 0, then we recover the original procedure. This version can be used in an episodic context as well as for infinite-horizon tasks like online tuning.

Model Parameters

Before applying the batch training algorithm to the agent, we need to identify the parameters that are most likely to affect the performance of the agent. These parameters are the focus of the tuning that takes place in the next section.

- Maximum depth (integer): Maximum number of successive nodes in a decision tree.
- Number of estimators (integer): The number of trees in the XGBoost ensemble.
- Interaction constraint (boolean): Allows multiple features to be used in succession as decision features.
- Gamma (positive real number): Specifies the minimum loss reduction for leaf splitting.

- Eta (real number between 0 and 1): Used as a learning rate for feature weight shrinkage.

The maximum depth and the number of estimators increase the complexity of the model and may encourage overfitting of the training data. Eta and gamma, on the other hand, are used in the tree-building process as regularization parameters and have the opposite impact. Lastly, constraining the interaction of features can make the final model more interpretable, but runs the risk of underfitting the data (Chen et al., 2016).

Environment Parameters

On top of the model parameters, the environment comes with its own set of parameters that need to be adjusted on the task. They have been listed below along with a short description.

- Number of epochs: Group of episodes during which only data collection is active.
- Number of episodes: Number of episodes in an epoch. An episode corresponds to a new array that has to be explored by the agent.
- Reuse best model: Specifies whether or not the agent should use the latest trained policy as its current policy or the best policy obtained until now instead.
- Epsilon: The proportion of actions that are exploratory in the ϵ-greedy policy.
- Epsilon initialization, Epsilon floor, epsilon decay: These parameters determine the scheduling mechanism used for epsilon. Epsilon starts at its initialization value and steadily decays until it reaches its minimum value. It then stays there until the end of an epoch.
- Discount factor: The discount factor has an impact on learning in infinite-horizon tasks as we will see in the next chapter. In the case of episodic tasks, however, gamma can be set to 1.

3.2 Experimental Results

3.2.1 Results

After running the parameter exploration procedure, the best agent named TunedXGB obtains a consistent score of 3.8 on the binary search task. The score is equivalent to an average episode length of 2.2. In other words, TunedXGB needs on average one guess less than the original heuristic to find the item in an array of size 15. When using the same configuration to train another agent

Characteristics	Agent1 (TunedXGB)	Agent2	Agent3
Number of runs	5	5	5
Number of episodes	350	350	350
Average reward[2]	4.58 (3.81)	3.80 (3.77)	3.78 (3.69)
Max Depth	3	4	4
Eta	0.3	0.4	0.4
Reuse best model	False	False	False
Interaction constraint	Off	Off	Off
Number of estimators	50	7	7

Table 3.1: Top three agents trained on arrays of size 15. The best agent, TunedXGB, cuts down the traversal time by one when compared to the midpoint heuristic.

Pivot selection method	Score	Average episode length
TunedXGB	3.81	2.19
Binary Search	2.75	3.26
Random pivot	2.10	3.90

Table 3.2: Table comparing TunedXGB with 2 baselines.

TunedXGB127 this time on arrays of size 127, we get a score of (only) 1.6. This is equivalent to finding an item 1.5 steps earlier than binary search. The relatively poor performance gives reason to believe that an agent specifically tuned on large arrays could outperform TunedXGB127.

During the tuning procedure of XGBoost, the vast majority of configurations have led to agents that score between 2.1 and 3.5, not significantly better than the original heuristic. Table 3.2.1 reports the configuration found for the top three agents two of which have the same configuration.

[2]The exceptional result of 4.58 found during the parameter exploration for Agent1 would amount to having the agent select the correct item in only one try over nearly 350 different arrays. Unfortunately, after trying to replicate this unlikely outcome, the same agent scores a disappointingly lower, but more realistic, 3.81. Agent2 and Agent3 suffer from the same selection bias. After correction, they boast a score of 3.77 and 3.69. Agent1 seems to have been a particularly extreme outlier in this regard.

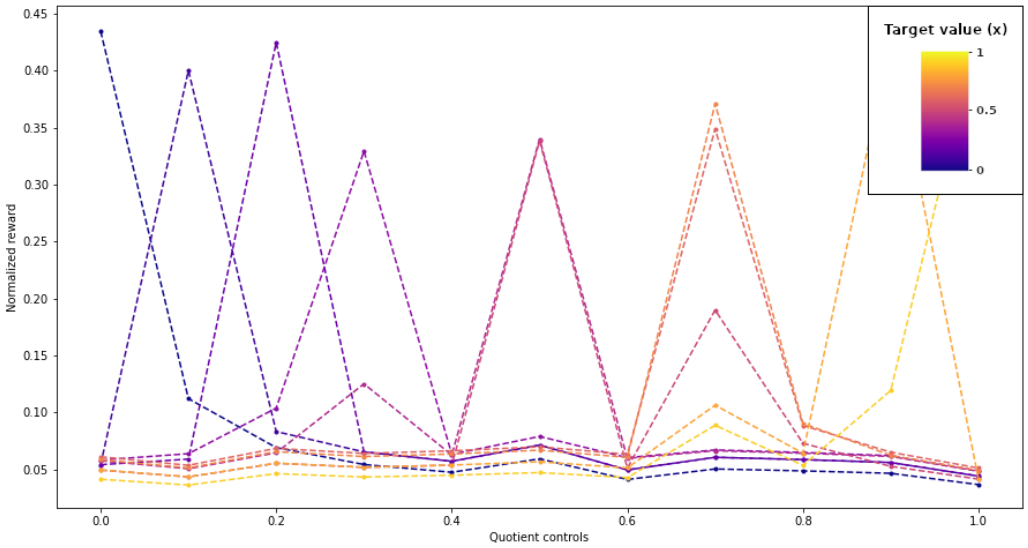

Figure 3.3: Q-values learned by TunedXGB when faced with a new episode. Each line represents the actions that are most likely to lead to the item. The lines are differentiated by color going from indigo when searching for item 0 to yellow for item 1. The agent seems to have learned an interpolating strategy at the beginning of an episode.

3.2.2 Model Interpretation

To garner additional insight into how TunedXGB outperforms ordinary heuristics, the next graphs show the activation of the model in three different states.

At the starting position, the agent correctly predicts that the pivot chosen should be proportional to the target object. This is consistent with the fact that the data is uniformly distributed and the agent is more likely to find item 0.2 at about two-tenths of the array.

Figure **??** and **??** show respectively the Q-values learned by the agent on the states (0, 0.5) and (0.5, 1). A similar phenomenon from the starting position is at play. The agent has learned the uniform distribution of the data but also takes implicitly into account that for it be in state (0, 0.5) it had had to take action 0.5 at the previous step. As a result, most predictions seem to be skewed towards the middle of the array, 0.5.

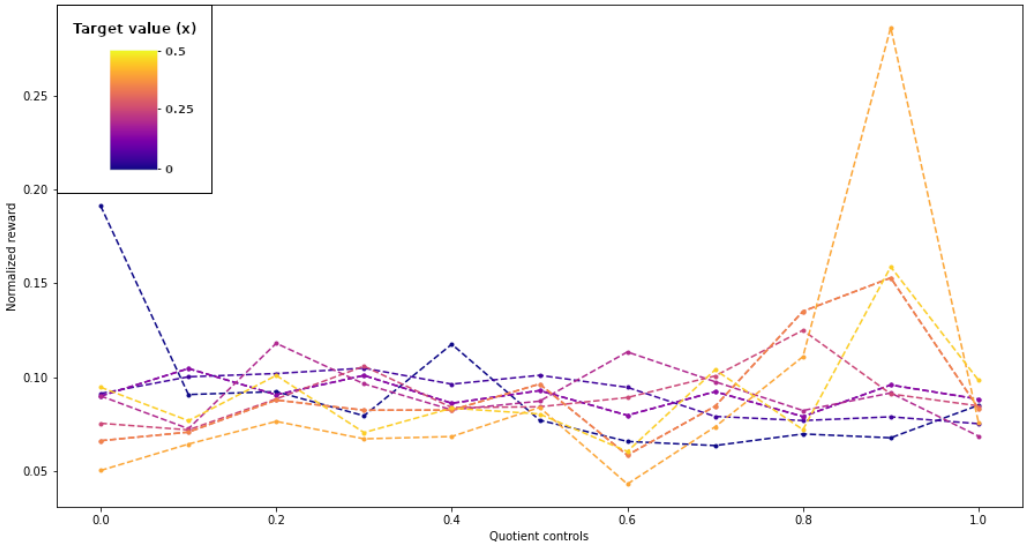

Figure 3.4: Q-values learned by TunedXGB in state [0, 0.5]. The activation lines exhibit a left skew that shows which the agent believes the item to be close to the middle of the array.

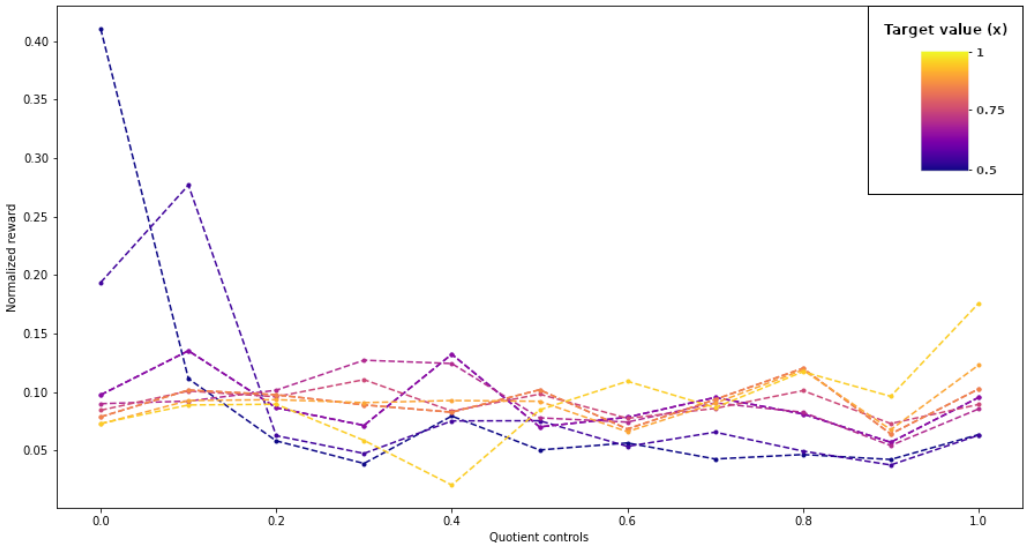

Figure 3.5: Q-values learned by TunedXGB in state [0.5, 1]. Likewise, the activation lines exhibit a right which shows that the agent believes the item to be closer to the middle.

3.3 Conclusion

In this chapter, we have analyzed a specific algorithmic problem and showed how a reinforcement learning agent could generate heuristics that outperform. After breaking down some of the theoretical guarantees of binary search, we have seen that decision trees require a special procedure to be trained in a reinforcement learning context. Afterward, we have looked at the parameters that have the biggest impact on the environment and the agents and how to tune them according to the task. We have seen that decision trees can serve both to approximate large space and also to generalize from examples when the state space is limited in size. With decision trees as function approximators for Q-learning, we have trained agents that noticeably exceed the base heuristic on the problem of sorting an array of sizes 15 and 127.

Naturally, the per-iteration cost of running the RL agent has to be factored into the cost-benefit analysis of implementing such a technique. Regardless, we have proven that the reinforcement learning alternative is easy to incorporate into existing code, spares the need to create a domain heuristic, and outperforms the heuristic given the right conditions. In the next chapter, we turn our attention to optimize hyperparameters in streaming pipelines. We reuse the findings of this chapter but generalize to online environments with multiple objective functions.

Chapter 4

Simulation

This chapter addresses the core problem of optimizing pipelines by tuning hyperparameters via stochastic control. Before describing the RL algorithms and showing our results, we describe the computing simulation which was used to run multiple pipelines in parallel.

4.1 Methodology and Benchmarking

4.1.1 Motivation for Building a Simulation

The purpose of the simulation is to validate the workability of reinforcement learning (RL) for online hyperparameter tuning before running the RL system on a real streaming platform. It demonstrates that agents can solve a dual optimization problem by namely reducing latency and maximizing throughput.

At the cost of it being harder to reproduce our results, we have chosen to develop a custom-made simulation because we wanted an environment that could be progressively extended in parallel to the agents from zero. To speed up the development cycle, the prototype environment should be less complex to experiment with than a full-fledged streaming framework. In the end, the simulation would be easier to benchmark and troubleshoot especially in the early stages of development.

Nonetheless, we were inspired by many existing elements from streaming platforms like the structure of the execution graphs and the sequential pipelines models, while adding our own features. In its latest form, the processing time response of workers is inversely proportional to the number of resources allocated to it. But the transfer function can be replaced by any monotonically decreasing function. This allowed us to measure the behavior of the agents in complex environments which would have been substantially harder to recreate within a streaming framework.

4.1.2 Processing Benchmarking

When measuring the performance of our system, we reuse the notions of processing-time latency, event-time latency, and throughput as performance metrics (Karimov et al., 2018). For our purposes, we chose the event-time latency and the throughput as the metrics to measure and optimize.

To begin with, the processing time represents the duration required by a worker to execute a unit of computation. The processing-time latency is what is usually thought of when speaking of latency. The event time builds on top of the processing-time latency and adds the time spent by the job inside the various queues of the message broker. The event-time latency is less preoccupied with the performance of individual workers than it is with the performance of the whole system. Finally, the throughput is obtained by counting the number of events that the system processes in a given period.

4.2 Solution Architecture

The architecture is composed of a user-facing API that mimics the original Java Stream API, the agent side, and the environment, a black box cluster simulation.

4.2.1 Optimizer API

In order to provide a frictionless road towards the adoption of the meta-heuristic, we based the RL API on the Java 8 Stream API. Java Stream objects can thus easily be upgraded to SmartStream. Where a hard-coded was previously generating a parameter, the user can now specify a bounded search space and let the optimizer handle the parameter.

```
// 0. Generate input data
// [...]

// 1. Specify the search spaces
SearchSpace nicenessSearchSpace  = new SearchSpace(minNiceness, maxNiceness, 0);
SearchSpace queueSizeSearchSpace = new SearchSpace(minQSize, maxQSize, 5);

// 2. Create the pipelines (execution graph)
final SmartStream<Double> stream1 = SmartStream.of(inputData)
    .map(x -> x + 1)
    .map(x -> x - 1);

final SmartStream<Double> stream2 = SmartStream.of(inputData)
    .map(x -> x + 2)
    .map(x -> x - 2);

// 3. Register them to the simulation
simulation.register(stream1, nicenessSearchSpace, queueSizeSearchSpace);
simulation.register(stream2, nicenessSearchSpace, queueSizeSearchSpace);

// Start the simulation
simulation.run();
```

4.2.2 Pipelines, Processing Times, and Drifts

The simulation cluster is made up of independent sequential pipelines. Each pipeline contains at least $n = 2$ workers connected by $n - 1$ queues for a total of $2n - 1$ agents. The pipeline model is a composition of functions where every worker applies a transformation on an incoming job before handing it off to the next queue. At the end of the pipeline, the job is simply discarded. As shown below, a minimal pipeline is a composition of two functions joined by one queue:

$$W \to Q \to W(\to Q \to W)^*$$

Every worker W_i has a built-in processing time $w_i \in \mathbb{R}^+$ which simulates actual work done by a computation unit. The values for w_i contribute directly to the processing-time latency defined in 4.1.2 and are an important part of what makes the environment dynamic. To make the environment more challenging for the agents, w_i are sampled from a Gaussian distribution $\forall i, w_i \sim \mathcal{N}(\mu_t^{(i)}, \sigma^2)$. Each $\mu_t^{(i)}$ is initialized at 100ms and drifts every m time step as follows $\mu_{t+1}^{(i)} \leftarrow \mu_t^{(i)} + \eta$ where $\eta \sim \mathcal{N}(m, s^2)$.

Agents are tasked with stabilizing the whole system and maximizing the objective function. Each agent A_i can draw from a common pool and allocate a discrete amount of resources r_i to worker W_i. The amount of resources r_i influences the **effective processing time** or wall-clock processing time

p_i of a worker. In the simulation, we make the simplifying assumption that all jobs are embarassingly parallelizable. This means that worker W_i will spend only $p_i = \frac{w_i}{r_i}$ time if granted r_i resources (eg. if $w_i \simeq 100\text{ms}, r_i = 10$ cores, then $p_i \simeq 10\text{ms}$).

4.2.3 Message Passing

The latest iteration of the architecture completely decouples the environment from the agents by using a message-passing system. On the left-hand side of Figure 4.2.3, the simulation creates an execution graph from the SmartStream pipelines. It instantiates the workers and queues with the workload and starts monitoring the performance of the system. Meanwhile, agents on the right-hand side are assigned a hyperparameter which they can optimize using many available algorithms. For the sake of illustration, the agents are coded in Python but the only requirement is for the agent to broadcast an action from a ZeroMQ endpoint regardless of the platform. In the middle of the diagram lies ZeroMQ, a message broker used for exchanging messages in an asynchronous way. The simulation broadcasts the state of the simulation as well as the rewards through ZeroMQ sockets. The Python agents then receive this information and respond with one of the three actions defined in the exploration model 4.3.2. From the perspective of a Python agent, the problem of controlling the simulation is a **contextual multi-armed (3-armed) bandit** problem.

4.2.4 Concurrency Considerations

Simulation

The simulation is a heavily multi-threaded construction. The implementation makes use of concurrency and thread-safe data structures in Java. Each worker is assigned its own thread to recreate a small-scale computation cluster. The primary data structure used is the *BlockingQueue<E>* from the Java concurrent library, which comes with thread-safe guarantees. Worker threads are designed to continually consume from queues and produce to new queues somewhat resembling the Kafka buffering system. The size of the queue can be updated by the agents following the locking guidelines. When the queue is full the worker enters a blocking state until the queue is vacated. The Java blocking mechanism has an inherent overhead which can be picked up in the jitters of the latency in Figure 4.6.2. Indeed, threads take on average 40ms to exit a blocking state from the moment they get notified. Not only is this overhead conspicuous, but the delay can sometimes compound to exhibit spikes in the latency. Nevertheless, the artifact is kept in the simulation and the agents have

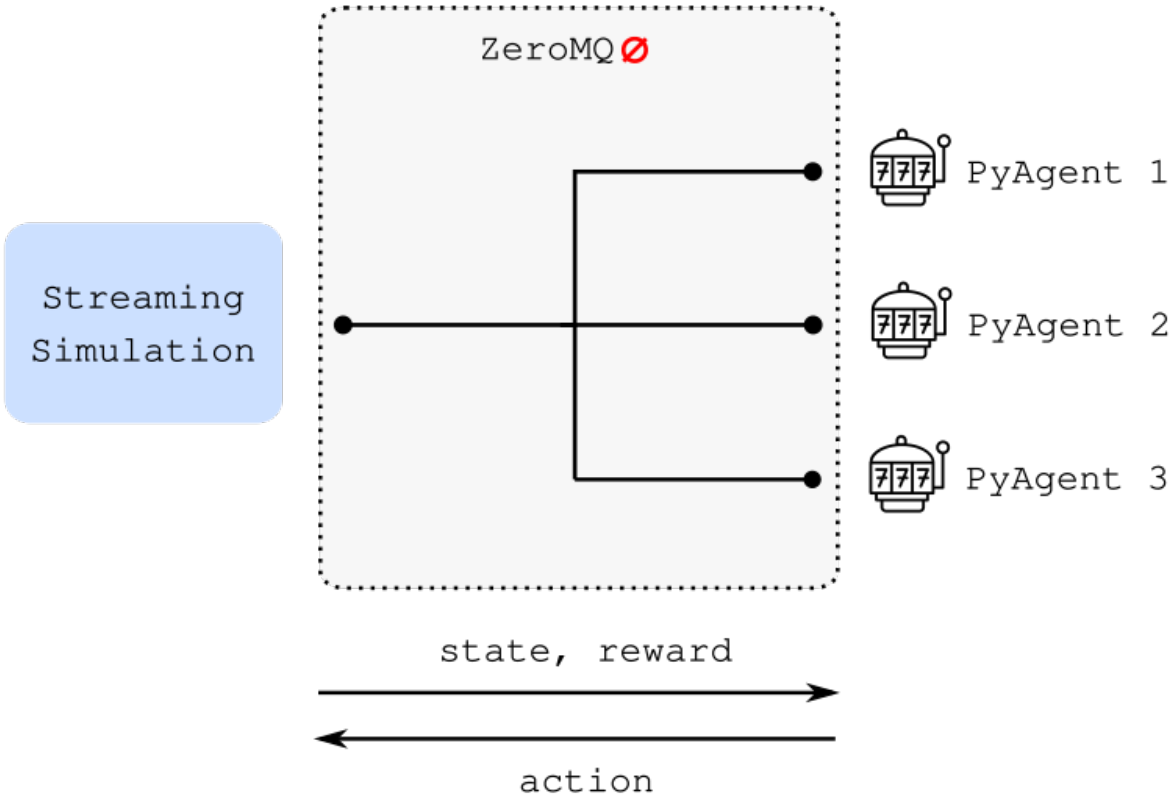

Figure 4.1: Architecture overview. The simulation is interconnected with the PyAgents thanks to ZeroMQ, a messaging library. The agents act asynchronously from the simulation and can be run on different points of a network if need be.

to work around it.

Agents

As for the agents, they too are acting asynchronously. If the agent does not compute its action in time then the object defaults to a NOOP operation and stays still. In practice, however, the inference time, as well as the training time of the agent, is significantly dominated by the clock of the simulation so that the agent never misses an action.

4.3 Reinforcement Learning Environment

4.3.1 Local Agents, Global Agents, and Omniscient Agents

When optimizing a complex system, there are many data sources to choose from. From the network activity, and CPU temperature, to engineered features, there is no limit to how much information one can pack in a state in the hopes that it improves the predictive power of an agent. It is somewhat hard to disentangle cause and effect when dealing with omniscient agents. That is why we have restricted the content of the state to the present value of all the parameters that are being optimized (eg. [size of queue1 = 5, size of queue2 = 6, resource1 = 10, resource2 = 20, ...]).

Restricting the amount of information an agent has access to gives rise to the distinction between local agents and global agents. Local agents are intended to know only the parameter they control and are decoupled from other agents. This decentralized mode of operation facilitates the process of scaling up a cluster. Global agents in contrast break the property of locality. They have access to all the parameters in the simulation and can thus learn more complex dependencies. In XGBoost, insulating agents from one another can be done by disabling the **feature interaction** option.

4.3.2 Exploration Model

The parameters of the functions we wish to optimize live in a simplified bounded integer space. As an example, the parameter can be the size of a queue in the cluster $q_i \in [\![1, 10]\!]$ or the niceness of a process $n_i \in [\![-20, 19]\!]$. The environment is built in such a way that the agent has three actions at its disposal which deterministically transitions to the next state:

- UP ($\uparrow$) increases the parameter by one: $(S_t, A_t = \uparrow, S_{t+1} = S_t + 1)$
- DOWN ($\downarrow$) decreases it by the same amount: $(S_t, A_t = \downarrow, S_{t+1} = S_t - 1)$

- NOOP ($\rightarrow$) stays put: (S_t, $A_t = \rightarrow$, $S_{t+1} = S_t$)

This exploration model is often used in the literature to explore ranges of parameters (Bu et al., 2009), (Xu et al., 2012), and (Peng & Zhang, 2017). When using this exploration model, two important constants come into play: the spatial resolution in the form of the step size and the temporal resolution in the form of the sampling rate. To the best of our knowledge, the literature does not mention a rigorous way to select them. We have therefore chosen an arbitrary step size of 1 and a sampling rate of 100ms.

Regularization and Jumping Process

The exploration model suffers from a shortcoming in that it is prone to get stuck in local optima. After running the simulation for a while, the agent may find itself 10 steps away from a significantly better region and yet only reach it with probability $(\frac{\epsilon}{3})^{10} \simeq 10^{-15}$ for a respectably sized $\epsilon = 0.1$. Techniques such as Upper Confidence Bounds (UCB) can encourage visiting sequences of states which have not been explored in a long time (Sutton & Barto, 1992).

In this simulation, we make use of an alternative exploration technique analogous to exploring starts. The technique is originally used to guarantee that states are visited infinitely often in Monte Carlo episodic tasks. This is achieved by choosing a random state as the first state, hence the name exploring starts (Sutton & Barto, 1992). In our case, we incorporate a **jumping process** that abruptly moves the state of the environment to a random location at regular intervals. This can be seen as a form of episodic re-initialization like exploring starts. More importantly, even though regularization is less of a problem in continuous tasks with dynamic environments the jumps act as a strong regularization mechanism. They push the agent to visit remote regions that would otherwise be left unexplored.

4.4 Optimization Objective: Latency and Throughput

4.4.1 Definitions

The latency and the throughput are the proxy metrics chosen to measure the performance of the cluster. A good analogy to understand the difference between the two metrics is that latency is individual productivity whereas throughput is organizational productivity. The throughput can be increased by adding additional resources to the cluster also known as **horizontal scaling**. This will

in turn increase the productivity of the whole system. On the other hand, increasing the buffering capacity is a good way to guarantee a higher throughput at the cost however of individual latency. In our application, the throughput is the number one priority, but a certain degree of responsiveness as manifested by low latency is desirable too.

The coupled optimization is a good opportunity to solve a more general reinforcement learning problem with multiple reward components. As we will see, this approach allows us to quantify the administrator's preference towards latency and throughput as a linear combination.

Based on the definition of Section 4.1.2, the metrics were measured as follows.

- **Throughput** is defined as the number of jobs completed over a certain time window. It is measured by taking the terminal workers and adding up their individual throughput. A job can represent any succession of operations on bounded input data.
- **Latency** is measured on a job-per-job basis. Every pipeline has a process that tracks the latency by computing the difference between completion time and the creation time of the last job.

4.4.2 Intuition Behind the Optimal Learned Behavior

In this section, we anticipate the training of the agents by giving the intuition behind the learned behavior. In the special case of a single pipeline $W_1 \to Q_1 \to W_2$ composed of 3 agents controlling the resource allocation and the size of the queue, one can attempt to describe what the optimal policy could be. As a general rule of thumb, when downstream workers are slower than upstream ones (processing time of W_1 is lower than processing time of W_2: $w_1 < w_2$), the queue will get clogged up and jobs will suffer from high latency. The reverse is true, if $w_1 \geq w_2$ then there will be no waiting time for the jobs, and latency will be at its lowest.

The difficulty comes from the dynamic nature of the environment. The processing time of a worker not only drifts over time but depends on resources allocated to adjacent workers. Therefore, we can say that the agent should learn a policy that equalizes the effective processing time $\frac{w_i}{r_i}$ as best they can to ensure the highest possible throughput. If the ratios cannot be equalized for any reason, then large queues can act as a buffer and reduce downtime.

This pseudo-optimal policy can be summarized in a few conflicting bullet points:

1. Maintain an ascending order between successive processing times of a pipeline while accounting for resource utilization. This leads to a **low latency** cluster and avoids **backpressure**.

2. If possible, the processing times should all be close to one another so as to **maximize the throughput**.

3. In a more volatile environment, large buffers can still be used to keep the **throughput high** at the expense of **more latency**.

4. The allocation of r_i needs to be homogeneous.

This informal understanding of the optimal policy is a good first approximation of what is going on under the hood of the agents. It was also used as a sanity check throughout the development to make sure the agents are behaving reasonably.

4.5 Convex Optimization

We can now tackle the problem of convex optimization and how to balance rewards. There are three main ways to solve a multi-objective optimization problem in reinforcement learning. The first solution is to convert the multiple objectives into a single-objective optimization problem also known as a **scalarized problem**. Since XGBoost cannot in its raw form predict convex hulls, scalarizing was the only viable option to train the gradient boosted machine. The second solution is often used in environments with hierarchical agents where possibly adversarial rewards are received from composite agents. Specialized solutions can be used to solve this problem but they do not apply to our problem (Shelton, 2006). The third and last solution is to keep the rewards separated and let a human administrator or operator define the subjective preference vector to combine the rewards. The operator can define a high-level strategy and let the algorithm implement the details. The operator explicitly quantifies how many units of latency are worth a unit of throughput and can change the policy on the basis of need. This solution along with the first one is explored in the next sections.

4.5.1 Problem Statement

Markov Decision Processes can be generalized to yield rewards with n components: $\vec{r} = \left[r_1, r_2, \ldots r_n\right]^\top$. When there is a need to rank a set of reward vectors to know for example which action to select, we

can project the vectors onto the number line. This is done by using a preference vector $\vec{\alpha}$ such that $\sum_i \alpha_i = 1$ and the $r_i(\vec{\alpha}) = \vec{\alpha} \cdot \vec{r}$ can be ordered.

In our case, the optimization is done with regards to the dual rewards of latency and throughput. After every action, the environment is probed and gives out a reward $\vec{r} = \left[r_{latency}, r_{throughput}\right]^\top$. To simplify the notation, we use a preference scalar and we write $r = \alpha \cdot r_{latency} + (1-\alpha) \cdot r_{throughput}$.

4.5.2 Reward Shaping

Although conceptually correct, the process of maximizing the rewards described in the previous section **does not** lead to the desired solution. Indeed, we wish to maximize throughput but **minimize** the latency of the system. The way this is rectified is by taking the inverse of the latency and scaling it by an amount that makes it comparable to the throughput. Doing so is a very common procedure in the literature called reward shaping. In many cases, properly scaling rewards can improve the overall performance.

4.5.3 Dual Optimization

Learning All Optimal Policies with Multiple Criteria describes a method to generalize Q-learning to multi-objective problems that we call $\mathring{Q}$-learning (Barrett & Narayanan, 2008).

Exploration - Exploitation Revisited

This method introduces a new exploration-exploitation trade-off. Previously, the compromise was between greedily choosing the actions that are known to lead to the highest reward or choosing suboptimal ones with the hope of discovering a better reward. Now that the rewards are parametrized by a preference vector, the question becomes should the agent explicitly follow the preference vector or anticipate changes in $\vec{\alpha}$ and learn a more general solution. Even though one could conceive of a δ-ϵ-policy where a δ proportion of actions is allocated to learning a target policy for different $\vec{\alpha}$, and an ϵ proportion is allocated to finding states that have better values than current ones, this question remains largely unaddressed in the following sections.

Difference Between Off-Policy and On-Policy Algorithms

With off-policy algorithms, the agent learns a target policy but follows a different policy called the behavior policy. The target and behavior policy coincide however with on-policy algorithms.

This difference can be illustrated by comparing the on-policy algorithm SARSA with its off-policy counterpart, Q-learning. For both algorithms, the behavior policy is an epsilon greedy policy, ie. the agent chooses the best action with probability $1 - \epsilon$ and a random action with probability ϵ. SARSA bases its learning on the behavior policy and thus accounts for the fact that random actions are taken with probability epsilon. The opposite is true for Q-learning. It evaluates the state as if the optimal greedy policy was used all along (see Figure A in the annex for a concrete example). Even though off-policy can lead to unexpected behaviors like the cliff-walking examples, most off-policy are used to improve exploration by following multiple policies at the same time. There is no on-policy alternative to $\mathring{Q}$-learning because there is no single policy to begin with.

The comparison between SARSA and Q-learning sheds some light on what is happening with $\mathring{Q}$-learning algorithm. The agent learns a general target policy π by filling its table with $\mathring{Q}$-values. When necessary, the convex hulls can be evaluated at a preference α' to recover the behavior policy $\pi(\alpha')$. One could hypothetically try to first gather as much information by filling the Q-table and only then specify an α to exploit this knowledge. But this approach suffers from two problems. The first is that a maximally explorative strategy that does not depend on α is suboptimal because it fails to take advantage of information conveyed by the reward signal (Sutton & Barto, 1992). This is especially relevant when the state space is large and selecting actions at random amounts to finding a needle in a haystack. The second problem is that for any non-stationary environment such as ours, the optimal is a moving target that changes all the time. For these reasons, Barett et al. sidestep the exploration issue entirely. The agent follows an epsilon-greedy behavior policy based on the Q-values evaluated at $\vec{\alpha}$ and nothing else.

4.5.4 $\mathring{Q}$-learning

Training Q-learning with Convex Hull

As described in Section 1.3.4, Q-learning is a reinforcement learning algorithm which in its basic form uses a table of action values to guide its policy updated through one-step backups:

$$Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a'} Q(s',a') \tag{4.1}$$

If we want to use the same backup on convex hulls, we need to define four operations. The addition between a vector and convex hull, the addition between convex hulls, the maximum operation

on convex hulls, and the operation of scaling a convex hull by a scalar (Barrett & Narayanan, 2008).

- The addition between vector and convex hulls is used to add the reward to the discounted value of future states. It can be defined as follows:

$$\vec{v} + \mathring{A} = \{\vec{v} + \vec{a} \mid \vec{a} \in \mathring{A}\}$$

- The Minkowski addition is the extension of the addition for scalars. It is used to apply the learning rate $(1 - \alpha)\mathring{Q} +_m \alpha\mathring{Q}$:

$$\mathring{A} +_m \mathring{B} = \{a + b \mid a \in \mathring{A}, b \in \mathring{B}\}$$

- The maximum operation of $\mathring{Q}$ is used to bootstrap the value of the state by assuming that the agent follows the action with the maximum value. Initially, the maximum always corresponded to a single state. Now that there are multiple objectives, one state can be the maximum with respect to one objective and another state can maximize a different objective combination. Whatever the case may be, maximizing $\mathring{Q}$ boils down to building a state convex hull from the individual action convex hulls (Barrett & Narayanan, 2008).

$$\max_a Q(s, a) = \bigcup_a \{\mathring{Q}(s, a)\}$$

- Scaling a convex hull by a discount factor reflects that $\mathring{Q}$-values are less desirable in the future. The scaling operation is defined as follows:

$$\gamma\mathring{A} = \{\gamma\vec{a} \mid \vec{a} \in \mathring{A}\}$$

We are now ready to define the backup for $\mathring{Q} - learning$, which is at the center of the algorithm.

$$\mathring{Q}(s, a) \overset{\alpha}{\Leftarrow} \vec{r} + \gamma \,\mathrm{hull} \bigcup_{a'} \mathring{Q}(s', a') \tag{4.2}$$

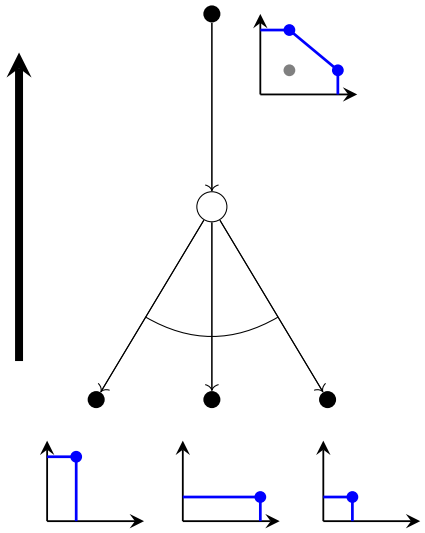

Figure 4.2: $\mathring{Q}$-learning backup diagram. Convex hulls of latter actions are backed up towards earlier states. They are then assembled into new convex hulls where only dominating rewards in blue are kept.

Recovering the Behavior Policy

At this stage, we a have precise way of capturing multi-dimensional values of states in a table with convex hulls. The last step is to determine how to create a policy from the convex hulls. To find the best action in state s knowing the values $\mathring{Q}(s,a)$, we need to order the values. The way this is done is by evaluating the $\mathring{Q}$-values at a preference vector $\vec{a}$. Afterwards the best action is selected as such: $a^* = \arg\max_a \vec{\alpha} \cdot r(a)$.

The final difference to notice is that the algorithm described by Barrett et al. assumes the knowledge of an environment MDP to compute an expectation in the backup. As such it falls in the category of dynamic programming (DP) algorithms under the name of $\mathring{Q}$-value iteration (see Figure A). However, we have no knowledge of the dynamics of the environments. That is why we have adapted their algorithm to learn from sampled transitions, which we call $\mathring{Q}$-learning. Algorithm 4.5.4 is the definitive description of $\mathring{Q}$-learning used by our agents.

Algorithm 3 $\mathring{Q}$-learning

for episode $= 1, M$ **do**
  Initialize s
  for step $= 0, T-1$ **do**
    Choose a from $\mathcal{A}(s)$ using the ϵ-greedy policy
    Observe r, s'
    $\mathring{Q}(s,a) \overset{\alpha}{\leftarrow} \vec{r} + \gamma \text{ hull}\{\bigcup_{a' \in \mathcal{A}(s')} \mathring{Q}(s',a')\}$
    $s \leftarrow s'$
  end for
end for

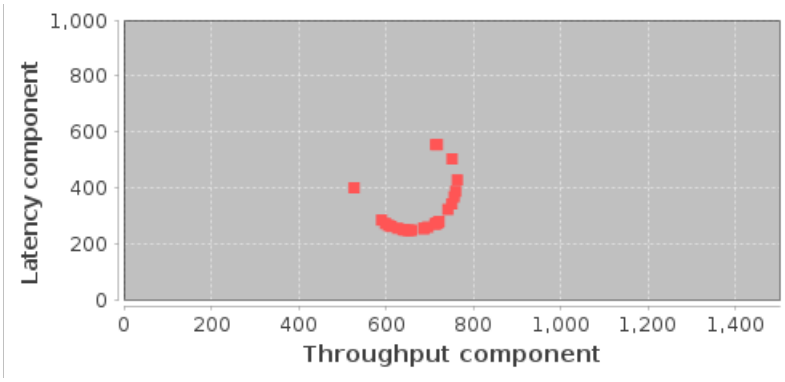

Figure 4.3: Screenshot of the JavaFX dashboard — A convex hull in action.

4.6 Experimental Results

Some experiments like modifying the preference vector and seeing if the agent adapts to the new operator policy are hard to quantify. Others like comparing the performance of XGBoost with a fixed preference vector with a $\mathring{Q} - learner$ do not lend themselves to meaningful comparison. The results have therefore been divided into two categories. The first is a set of qualitative results or observations that can be easily reproduced but are hard to quantify. The second is a set of quantitative results, which demonstrate the impact of tuning various parameters on the performance of the agent and identifies a top performer on the task.

For the purpose of creating a controlled environment, the drift is activated every 10 seconds. The same goes for the jumping process. The discount factor gamma was set to 0.9 and the starting preference value α at 0.5.

4.6.1 Qualitative Results

Listed below are some of the parameters that have a strong impact on the performance of the agents as well as some noteworthy observations.

- Local/global: A local agent has access to the state of the object it is controlling whereas a global has access to the state of the entire simulation. After testing both local and global agents, we see substantial improvements across the board when using a global agent. This can be attributed to the fact that each parameter is highly dependent on the value of the other parameters as seen in 4.4.2. The agents do not necessarily converge much faster but they are stabler than local agents because of the additional knowledge of the state. The interaction of all the features is essential to solving this task and is used by default in all the agents.

- Optimistic initialization: Initializing the Q-table with high action values is a known way to encourage exploration. The correct initialization value is highly dependent on the scale of the rewards and the choice of the discount factor (Rashid, Peng, Böhmer, & Whiteson, 2020). With a discount factor of 0.9, the best initialization value after several runs was found to be 720.

- Soft-greedy policy: The softmax function is used in the soft-greedy policy to select the best action according to its value. Unlike epsilon-greedy, the soft-greedy policy is stochastic and will occasionally select a sub-optimal action. Softmax has shown to be useful in two scenarios. First, when the Q-table is initialized and all values are identical, softmax can help break arbitrary ties — especially when optimistic initialization is not used. Second, softmax plays well with decision trees as they tend to produce identical outcomes and softmax can again break ties.

- Jumping process: After running the simulation for a few minutes, the agents learn the optimal policy and converge towards a stable region. Manually pushing the agents off course, or waiting for the jumping process, momentarily decreases the performance of the system. But the agents revert to the learned policy in a matter of tens of seconds.

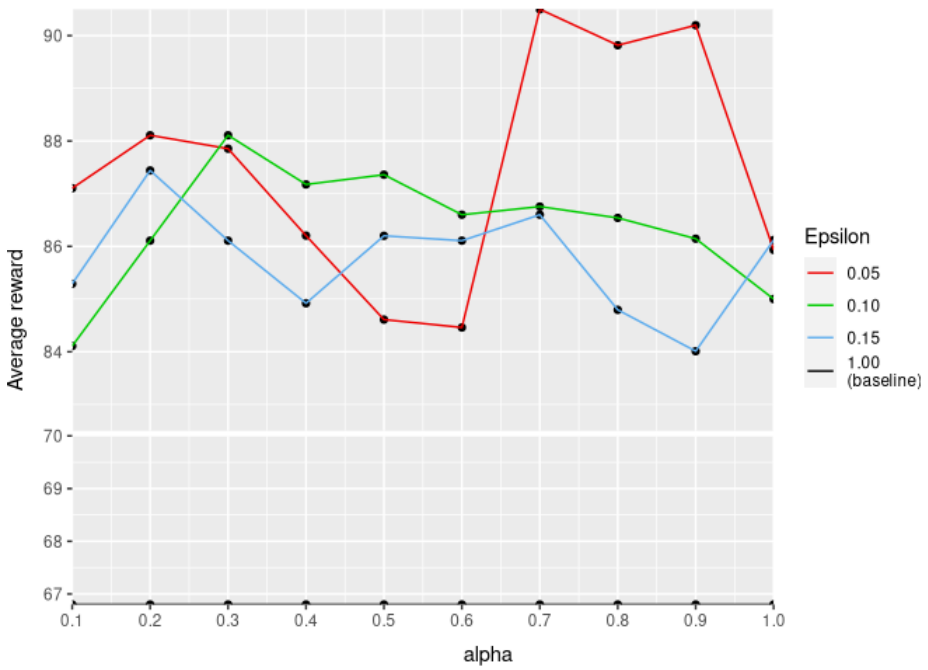

Figure 4.4: Average reward of Q-learners as a function of the learning rate parametrized by ϵ.

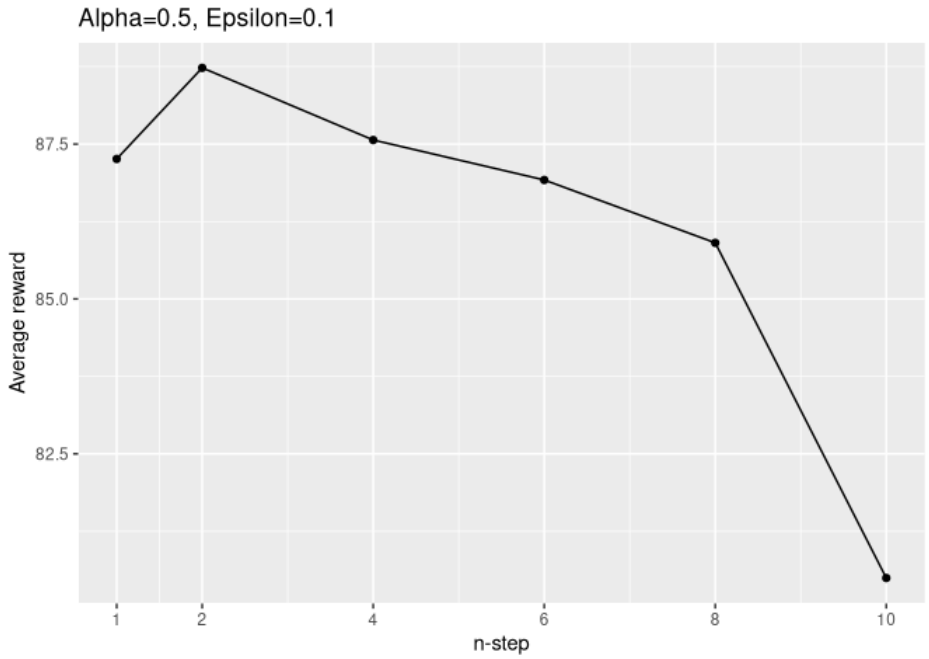

Figure 4.5: Average reward as a function of the number of steps. The graph exhibits its usual concave down shape.

4.6.2 Quantitative Results

In addition to the previous observations, various agents have been scored and ranked according to the average reward obtained. Each data point is the result of averaging 5 runs by the agent on the simulation. A run lasts approximately 8 minutes, enough for 2000 actions at a rate of 4 actions per second. We can see in Figure 4.6.2 the impact of both the learning rate and the exploration factor on the performance of a vanilla Q-learner. The green line associated with an epsilon 0.1 displays the expected shape of a concave down meaning that an intermediate value of the learning rate will produce the best performance for the agent.

In this case, an alpha of 0.3 reached the performance of 88 with an epsilon of 0.1. The relation is less clear when epsilon is equal to 0.05 or 0.15. Nonetheless, it seems that an epsilon of 0.05 and a learning rate in between 0.7 and 0.9 is the most suited combination for our task reaching an average reward of 90 — 23 points better than a random agent. Finally, at the bottom of the graph, we can see an agent with 100% exploratory actions used as a comparative baseline with a score o 66.8.

Figure 4.6.2 confirms that intermediate values of the number of steps are better than either the usual one-step bootstrapping or longer returns. In fact, an n-step value of 2 sees the maximum performance followed by a steady decline and a sharp drop in performance.

4.7 Further Improvements

Afterstate

Ignoring the jumping process, the simulation is deterministic in the sense that the action UP always increases the parameter by one. When the transitions in the MDP are well-behaved, the (state, action) formulation in Q-learning is redundant. Afterstates essentially contract the values ($S_t = i$, $A_t = \text{NOOP}$), ($S_t = i - 1$, $A_t = \text{UP}$), ($S_t = i + 1$, $A_t = \text{DOWN}$) into a single representation namely $S_{t+1} = i$ (Sutton & Barto, 1992). The implementation of afterstates can ease memory requirements by a factor of 3 and improve the learning of tabular solutions. Although the interaction with the jumping process is undetermined, afterstates could be good way to squeeze out additional performance from the agents.

Figure 4.6: Grafana screenshot n°1 — Three objects being actuated by the agents. The upper graph corresponds to the size of the connecting queue. It has only a minimal impact on the objective function. The lower graph shows a sustainable allocation of resources to the workers.

Figure 4.7: Grafana screenshot n°2 — The response of the environment to the agents. The upper graph displays how the environment converts niceness to actual worker performance. The lower graph shows the reward function explicitly optimized by the agents.

Parametric Approximation of Convex Hulls

Function approximators are involved at various stages of the reinforcement learning problem. They usually come in when the state space or action space becomes too large to be tractable via dynamic programming methods. Another form of approximation rarely mentioned in the literature comes from the formulation of multi-objective optimization. We have seen how the union operation and the Minkowski addition replace the maximum operation and addition on the convex hull, but that begs the question. Can we apply the same approximation techniques viewed in one-dimensional Q-learning to multi-dimensional Q-learning? Can we find a parametric expression for convex hulls that can generalize over multiple states, and if so would that form improve the performance of the agents on the reinforcement learning problem?

Deep Reinforcement Learning

Although decision trees have very practical advantages, most recent advances in the field of reinforcement learning have stemmed from the use of deep neural architectures like DDQN **??**. A powerful method to create meaningful features out of various sources of data coming from a streaming architecture could be neural networks. Neural networks come with their own set of engineering problems and we leave this for further works.

4.8 Conclusion

In this chapter, we have analyzed the software stack and decisions involved in building a streaming simulation. The simulation comprises a user-facing API, an environment for running execution graphs, and a set of agents to optimize the hyperparameters. The latest iteration of the simulation decouples the environment from the agent in a flexible and asynchronous way. A message-passing library is used to piece together the different components of the simulation. Moreover, taking a closer look at the environment has helped us articulate a pseudo-optimal policy that is used as a comparison point. The latter part of the chapter has introduced the problem of multi-objective optimization in the context of reinforcement learning along with the generalization of Q-learning to its multi-dimensional counterpart $\mathring{Q}$-learning. Subsequently, we have listed the major parameters that could be tuned in the environment and the agent. We have quantified the impact of these parameters on the performance of the agent and identified the best-performing agent on the task.

This chapter concludes my thesis on hyperparameter tuning of complex systems. The chapter has built on top of the previous chapter and has shown how agents can interact in an online environment to tune a large set of hyperparameters. We have not only created a heuristic enhancer that can be brought to bear on algorithmic problems but also a technique that can be used as a general gradient-free optimizer in online settings. The optimizer is able to create heuristics starting from scratch through the use of reinforcement learning agents coupled with decision tree ensembles.

The thesis represents the first step towards better framing the problem of autonomic computing from both sides. We have identified several unique problems in the setup, training, and inference of online agents, and have either solved them or pointed in the direction of literature that could solve them. Aside from the practical advantages of the reinforcement learning approach, we have seen that machine learning is competitive and consistently outperforms simple baselines albeit at the cost of diligent model tuning. Modified versions of Q-learning can be applied to a variety of tasks. We hope to have exposed some of the more promising applications of reinforcement learning in autonomic computing and have contributed to the realization of that objective.

References

Barrett, L., & Narayanan, S. (2008). Learning all optimal policies with multiple criteria. In *Proceedings of the 25th international conference on Machine learning - ICML '08* (pp. 41–47). Helsinki, Finland: ACM Press. Retrieved 2021-05-12, from http://portal.acm.org/citation.cfm?doid=1390156.1390162 (00154) doi: 10.1145/1390156.1390162

Bu, X., Rao, J., & Xu, C.-Z. (2009, June). A Reinforcement Learning Approach to Online Web Systems Auto-configuration. In *2009 29th IEEE International Conference on Distributed Computing Systems* (pp. 2–11). Montreal, Quebec, Canada: IEEE. Retrieved 2021-06-11, from http://ieeexplore.ieee.org/document/5158403/ (00101) doi: 10.1109/ICDCS.2009.76

Carbune, V., Coppey, T., Daryin, A., Deselaers, T., Sarda, N., & Yagnik, J. (2018). Predicted variables in programming.

Carbune, V., Coppey, T., Daryin, A., Deselaers, T., Sarda, N., & Yagnik, J. (2019, June). SmartChoices: Hybridizing Programming and Machine Learning. *arXiv:1810.00619 [cs, stat]*. Retrieved 2021-01-30, from http://arxiv.org/abs/1810.00619 (arXiv: 1810.00619)

Dulac-Arnold, G., Mankowitz, D., & Hester, T. (2019, April). Challenges of Real-World Reinforcement Learning. *arXiv:1904.12901 [cs, stat]*. Retrieved 2021-08-12, from http://arxiv.org/abs/1904.12901 (00209 arXiv: 1904.12901)

Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., & Markl, V. (2018, April). Benchmarking Distributed Stream Data Processing Systems. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 1507–1518. Retrieved 2021-06-11, from http://arxiv.org/abs/1802.08496 (00127 arXiv: 1802.08496) doi: 10.1109/ICDE.2018.00169

Levine, S., & Abbeel, P. (2014). Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics. , 11. (00432)

Li, H., & Venugopal, S. (2011). Using reinforcement learning for controlling an elastic web application hosting platform. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11* (p. 205). Karlsruhe, Germany: ACM Press. Retrieved 2021-08-15, from http://portal.acm.org/citation.cfm?doid=1998582.1998630 (00054) doi: 10.1145/1998582.1998630

Li, K., & Malik, J. (2016, June). Learning to Optimize. *arXiv:1606.01885 [cs, math, stat]*. Retrieved 2021-07-07, from http://arxiv.org/abs/1606.01885 (00320 arXiv: 1606.01885)

Liu, X., Li, J.-H., & Duan, S.-Y. (2012). Configuration optimization of hadoop based on chaos pso algorithm. *Computer Engineering*, *38*(11), 186–188.

Peng, C., & Zhang, C. (2017). A reinforcement learning approach to map reduce auto-configuration under networked environment. , 6. (00006)

Pyeatt, L. D., & Howe, A. E. (1998). Decision Tree Function Approximation in Reinforcement Learning. , 9. (00002)

Rashid, T., Peng, B., Böhmer, W., & Whiteson, S. (2020). Optimistic exploration even with a pessimistic initialisation. *CoRR*, *abs/2002.12174*. Retrieved from https://arxiv.org/abs/2002.12174

Shelton, C. R. (2006, January). *Balancing Multiple Sources of Reward in Reinforcement Learning:* (Tech. Rep.). Fort Belvoir, VA: Defense Technical Information Center. Retrieved 2021-04-30, from http://www.dtic.mil/docs/citations/ADA454702 doi: 10.21236/ADA454702

Sutton, R. S., & Barto, A. G. (1992). Reinforcement Learning: An Introduction. , 352.

Tesauro, G., Jong, N., Das, R., & Bennani, M. (2006). A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *2006 IEEE International Conference on Autonomic Computing* (pp. 65–73). Dublin, Ireland: IEEE. Retrieved 2021-06-11, from http://ieeexplore.ieee.org/document/1662383/ (00417) doi: 10.1109/ICAC.2006.1662383

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

Xu, C.-Z., Rao, J., & Bu, X. (2012, February). URL: A unified reinforcement learning approach for autonomic cloud management. *Journal of Parallel and Distributed Computing*, *72*(2), 95–105. Retrieved 2021-06-11, from https://linkinghub.elsevier.com/retrieve/pii/S0743731511001924 (00170) doi: 10.1016/j.jpdc.2011.10.003

Appendices

Appendix A

Additional material

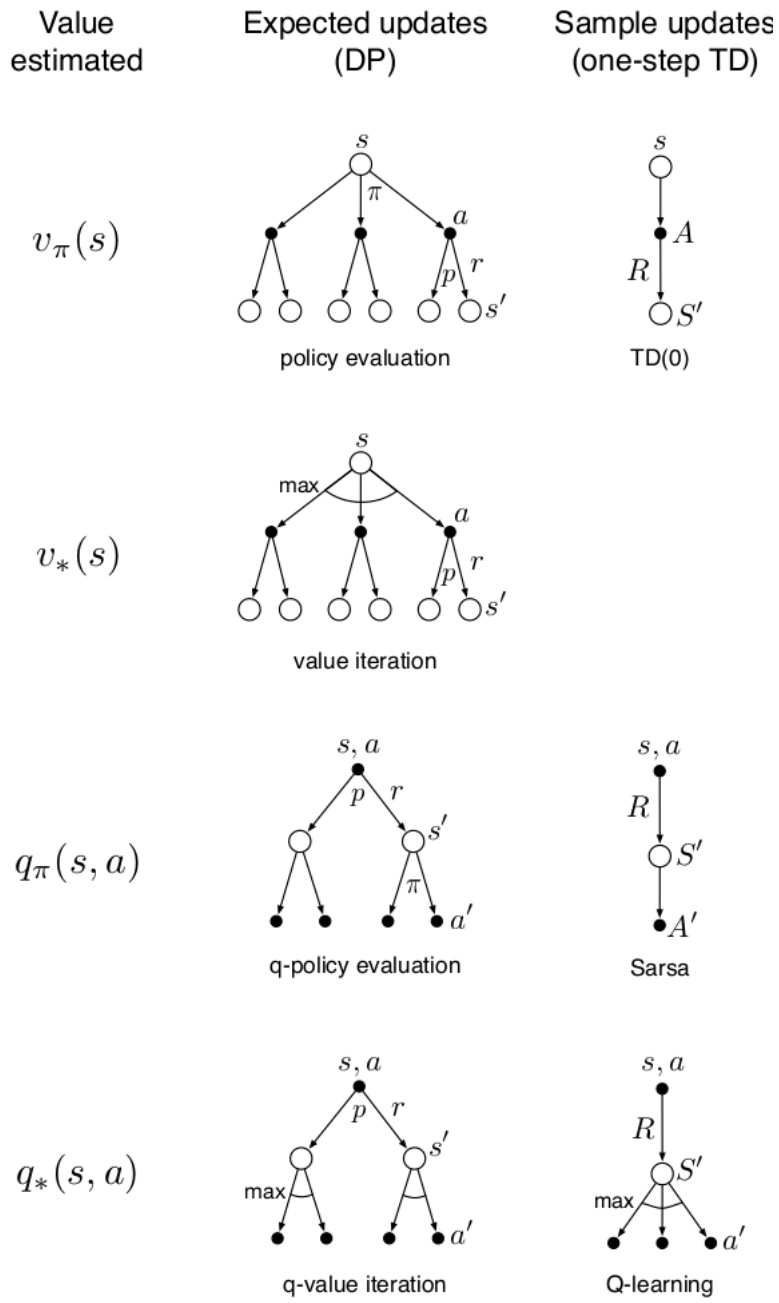

Figure A.1: Backup diagrams of popular RL algorithms. Large empty circles correspond to states while smaller black circles correspond to actions. In the case of Q-learning, the value of the topmost node $Q(s,a)$ is updated by reaching the first state S' and taking a peak at the maximum action value starting from that state. We get the familiar target $\delta = r + \gamma \max_{a'} Q(s',a')$. [Sutton and Barto, 1998]

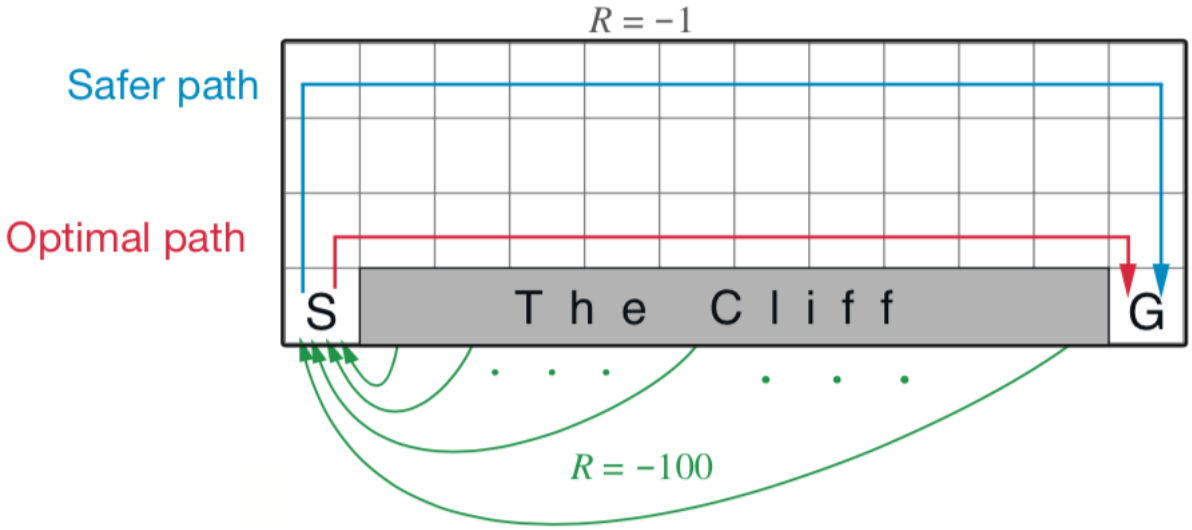

Figure A.2: Cliff-walking task illustrating the policies learned by on-policy SARSA (safer path) compared to off-policy Q-learning (optimal path). Unlike Q-learning, SARSA takes into consideration that an epsilon proportion of its actions are exploratory and may lead into the cliff. It thus takes the longer path. [Sutton and Barto, 1998]