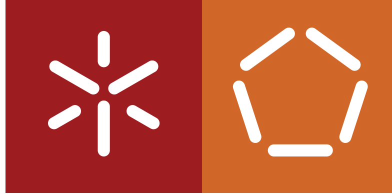
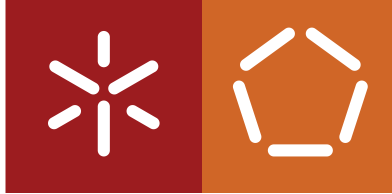


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Susana Vitória Sá Silva Marques

Autonomous Optimization for a Transactional Middleware

October 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Susana Vitória Sá Silva Marques

Autonomous Optimization for a Transactional Middleware

Master dissertation
Integrated Masters in Informatics Engineering

Dissertation supervised by
Doutor Fábio André Castanheira Luís Coelho
Professor Doutor José Orlando Roque Nascimento Pereira

October 2022

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:

CC BY
https://creativecommons.org/licenses/by/4.0/

ACKNOWLEDGEMENTS

This stage of my life was one of the most important ones and helped me grow in all aspects, personally, academically and professionally. I could count on the help of many people to make this happen. Here, I express my greatest thanks to all of those who helped me finish this chapter and who supported me this last year.

First of all, I would like to thank my advisor, Prof. Fábio André Coelho, who guided me throughout all the questions and challenges faced. His supervision, commitment and advice made my journey easier and allowed me to achieve the goals desired.

I would also like to thank my advisor, Prof. José Orlando Pereira, whose opinions and constructive criticism helped improve my work and move forward.

I must thank my family for helping me achieve my dreams and being there for me throughout all of the process. My mother for the unconditional encouragement and my father for the extraordinary comprehension and attention.

Finally, I want to say thanks to my boyfriend, Tomás, for all the patience he has shown, for all the technical help provided and for keeping me sane during this long journey.

I would also like to thank INESC TEC for this opportunity. This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

In the last few years, data management engines have become increasingly modular, separating some of its main layers, such as data storage and transactional management. The exposure of the transactional management component brings new challenges, in particular its correct configuration and tuning when running different workloads. In this sense, this dissertation focuses on the autonomous optimization of a particular transactional middleware, pH1, while keeping in mind the tuning of other similar systems.

It is becoming more and more important to develop algorithms that can automatically optimize these systems whose performance is heavily dependent on a proper configuration. The use of machine learning techniques for similar problems (database knob tuning) has become common in the literature [1, 2, 3, 4], especially in a black box perspective where it does not have visibility over particular details of the system.

Usually, these systems are located in realistic online environments, where workloads can change at different times. Even though there are numerous research projects for automatic knob tuning, these projects have not entirely addressed this problem and are mostly developed for offline training when the workloads remain static.

We propose OPAL as the component that when executing transactional workloads is able to dynamically adjust its configurations in an online environment with a continuous space. Our approach allows for online changes and uses reinforcement learning as a starting point taking into consideration tuning algorithms in continuous spaces, as is the case of DDPG [5].

KEYWORDS reinforcement learning, actor-critic methods, online environments, black box system.

RESUMO

Nos últimos anos, os motores de gestão de dados têm-se tornado cada vez mais modulares, separando algumas de suas camadas, principais, como o armazenamento de dados e a gestão transacional. A exposição do componente de gestão transacional traz novos desafios, em particular a sua correta configuração e ajuste durante a execução de diferentes cargas de trabalho. Neste sentido, esta dissertação foca-se na otimização autónoma de um *middleware* transacional específico, pH1, tendo em mente o ajuste de outros sistemas semelhantes.

É cada vez mais importante desenvolver algoritmos que possam ajustar automaticamente estes sistemas cujo desempenho depende muito de uma configuração adequada. A utilização de técnicas de *machine learning* para problemas semelhantes (ajuste de parâmetros em bases de dados) tem-se tornado comum na literatura [1, 2, 3, 4], sobretudo recorrendo a uma visão *black-box* onde não se tem visibilidade sobre detalhes particulares do sistema.

Normalmente, estes sistemas encontram-se em ambientes *online* realistas, onde as cargas de trabalho podem mudar em momentos diferentes. Apesar de existirem inúmeros projetos de pesquisa para o ajuste automático de parâmetros, estes projetos não abordam totalmente este problema e são desenvolvidos principalmente para treino *offline* quando as cargas de trabalho permanecem estáticas.

OPAL é proposto como o componente que ao executar cargas de trabalho transacionais é capaz de ajustar dinamicamente as suas configurações num ambiente *online* com um espaço contínuo. A nossa abordagem permite mudanças *online* e utiliza *reinforcement learning* como ponto de partida tendo em consideração algoritmos para ajuste de parâmetros em espaços contínuos, como é o caso do DDPG [5].

PALAVRAS-CHAVE aprendizagem por reforço, métodos ator-crítico, ambientes *online*, *black box* vision

CONTENTS

1 **Introduction** 1

1.1 The Problem 1

1.2 Main Aims and Contributions 2

1.3 Dissertation Structure 2

2 **State of the Art** 4

2.1 Machine Learning Spectrum 4

2.1.1 Supervised Learning 4

2.1.2 Unsupervised Learning 6

2.1.3 Neural Networks 6

2.2 Reinforcement Learning 8

2.2.1 Policy Optimization: Policy Gradient 11

2.2.2 Value Optimization: Q-Learning, DQN and DDQN 11

2.2.3 Actor-Critic: DDPG, TD3 and SAC 14

2.3 Machine Learning Tuning Systems 16

2.3.1 OtterTune 16

2.3.2 CDBTune 18

2.3.3 QTune 19

2.3.4 Multi-Model Tuning Algorithm 20

2.4 Transaction Management 21

2.4.1 pH1: A Transaction Middleware for NoSQL 22

2.4.2 Related Work on Transaction Management Systems 23

2.5 Discussion 24

3 **OPAL** 26

3.1 Architecture 26

3.2 Wrapper Module 28

3.2.1 Monitoring System and Tuning Parameter 28

3.3 Learning Module 29

3.3.1 Adapter Class / Custom Environment 30

3.3.2 Model Architecture 31

3.3.3 Decisions 32

4 **System Analysis and Performance Evaluation** 37

4.1 Preliminary Tests 37
4.1.1 Experimental Setup 37
4.1.2 Results 41
4.1.3 Discussion 44
4.2 pH1 Tests 46
4.2.1 Experimental Setup 46
4.2.2 Results 47
4.2.3 Discussion 51
5 Conclusion 53
5.1 Future Work 53

LIST OF FIGURES

Figure 1	ANN structure from [6].	7
Figure 2	Reinforcement learning simple procedure from [7].	8
Figure 3	Reinforcement learning functions.	9
Figure 4	Reinforcement learning agents.	10
Figure 5	Gradient descent.	11
Figure 6	Q-Learning from [8].	12
Figure 7	Deep Q-Learning from [8].	12
Figure 8	The principle of DQN algorithm from [9].	13
Figure 9	DDPG.	14
Figure 10	OtterTune machine learning pipeline from the original paper [1].	17
Figure 11	CDBTune system architecture from the original paper [2].	18
Figure 12	DS-DDPG method from the original paper [3].	19
Figure 13	Multi-Model database online tuning system [4].	20
Figure 14	Overview of a configuration with two pH1 instances from the original paper [10].	22
Figure 15	OPAL architecture.	27
Figure 16	Wrapper interconnections.	28
Figure 17	Learning module overview.	29
Figure 18	Model architecture.	32
Figure 19	Wrapper for simulations.	38
Figure 20	Function $y = x$.	38
Figure 21	pH1 latency variation when populating Cassandra.	39
Figure 22	Linear function with a noise coefficient of 0.2.	40
Figure 23	Function $y = \frac{sin(8 \cdot x)}{2} + x$.	40
Figure 24	Complex function with a noise coefficient of 0.2.	41
Figure 25	Performance of scenario A.	42
Figure 26	Performance of scenario B.	42
Figure 27	Performance of scenario C.	43
Figure 28	Performance of scenario D.	43
Figure 29	Comparison between different models.	45
Figure 30	Wrapper for pH1.	46
Figure 31	Throughput in a read workload.	47
Figure 32	Average read latency in a read workload in pH1 with and without OPAL.	48
Figure 33	Throughput in a mix workload (50% reads and 50% updates).	49
Figure 34	Average latency in a mix workload (50%reads and 50%updates).	49
Figure 35	Throughput in a mix workload with 5 threads.	50

Figure 36 Average latency in a mix workload with 5 threads. 50

LIST OF TABLES

Table 1 Mapping actor-critic algorithm to OPAL tuning. 29
Table 2 Set of elements that compose the state in the reinforcement learning process. 33
Table 3 Actor-critic hyperparameters (baseline). 33
Table 4 Hyperparameters tuned (improvement). 44
Table 5 Overview of the results obtained running a mix workload after an update workload. 51

1

INTRODUCTION

The highly-available and ubiquitous appeal of the *Cloud* currently steers a usage pattern where data is offloaded and processed in the *Cloud*, with the expectation that by 2025, 49% of all data in the world will be stored in public *Cloud* environments [11].

Providing continual availability and support for millions of users (some of the core properties of the cloud computing paradigm) has been a major business opportunity for cloud service providers. The high availability goal along with the competitiveness of storage systems drew an increasing number of businesses to migrate their systems into the *cloud*, thus reducing ownership and maintenance costs.

A new type of database systems arose to meet this demand. These are non-relational, typically just based on a key/value data model, and usually, they do not provide a complex structured query language (just simple *put* and *get* primitives) nor have transactional properties. are frequently known as NoSQL databases. A NoSQL database is expected to be highly available and scalable regardless of its unique data type or API. Key to these two characteristics are their inherent distributed design and weak consistent data replication. Recent attempts to provide transactional support to these systems have come at the cost of additional resources or restrictive constraints, making its efficient support extremely challenging.

Most components of transaction management systems in use today consist of a set of configurations that have a significant impact on the system performance. The optimal parameter tuning will depend, for example, on the workload the transaction management system receives. Properly configuring and tuning these parameters can often result in a better performance than what it would be achieved with the default configuration.

1.1 THE PROBLEM

The system administrator is often in charge of setting the values for configuration parameters.

There are two essential requirements the administrator need to have for parameter tuning. The first is general system tuning experience, as well as experience with the specific system, the hardware setup of the machine, and even the workload. The second is time. Trial and error are typically involved in this process, which takes time. Before applying the configurations to the actual production systems, it will probably also be necessary to utilize (and have) a test computer where the administrator can run tests. Even after all of this work, the configuration values may still fall well short of ideal, keeping in mind that the workload may dynamically change.

Setting optimal values for configuration variables is hard since there are a large number of variables and there are a lot of complex ways in which the values of one variable interact with the values of another. Moreover, if the value space for the variables is continuous, an exponential increase in the search space is also expected. This

can be seen in a transactional middleware that can be tuned in a way that improves the high overhead it contains compared with a NoSQL solution.

This overhead is due to the transactional properties in the system, in particular how the management of memory is conducted. The garbage collector execution time, for example, is a metric that can in fact minimize the harsh impact of transactions executions by cleaning the cache and the write-set queues from time to time. The difficulty comes not only from the fact that it is difficult to find the right time to do it but also as it is difficult to optimize a system composed of many others: the middleware itself, the Timestamp Oracle, the database Cassandra (although jointly used by the middleware).

1.2 MAIN AIMS AND CONTRIBUTIONS

This dissertation introduces, designs and details an automatic parameter tuning system, geared for transactional middleware systems, more specifically, adopting a black box approach, while employing an online environment.

OPAL, implements different reinforcement learning algorithms (used mostly in continuous spaces) so that the user can choose which one is the best to optimize the transactional system at hand or even compare different algorithms and optimizations.

The proposed solution improves the systems decision-making process, making it more dynamic and adaptive. It offers a flexible and expandable architecture supporting new and distinct systems due to its black box nature.

For systems that can not be tuned offline, OPAL is structured in a way that allows for the values to be adjusted dynamically while the model keeps learning and training online. This means that values are updated in real time, according to changes in the workload as long as Opal is running.

OPAL is evaluated in pH1 with 4 different benchmarks: read and mix workloads, different number of threads and multiple workloads running one after the other. It can achieve a better performance up to 15% when compared against pH1 by itself, tuning only one parameter. For the certainty that the models used in OPAL are well configured and implemented, some preliminary tests were also built, where the main goal was to optimize mathematical functions that could simulate the environment of the system being optimized.

1.3 DISSERTATION STRUCTURE

This document holds 5 chapters. The first one is an introduction to the dissertation, stating the problem, motivation and objectives. In Chapter 2, a machine learning spectrum with emphasis on reinforcement learning and the related work on tuning systems will be extensively described and explained. A background about transactional systems will be presented enabling a deep insight into the application case (pH1) and providing a solid understanding of the problem at hands before the introduction to the proposed solution. The proposed solution is elaborated in Chapter 3 by first expressing the adopted system architecture and going in-depth into the developed component systems, explaining how the connection between all the components is made and the decisions made for the implementation of the machine learning models. In chapter 4, the tests that were made to analyze this system are explained, along with how and why they were made and the performance obtained. Finally at the end of the chapter there is a discussion on the results this dissertation produced. The last chapter finishes

this document with an overview of what was done, the conclusions obtained from the results, and how this work could be further developed in the future.

2

STATE OF THE ART

2.1 MACHINE LEARNING SPECTRUM

The world now has access to a broad range of machine learning techniques that enable out-of-the-box solutions for the most diverse sectors, ranging from robotics to medicine and many more.

Tom Mitchell [12] sets the definition of machine learning in the following way: *"a computer program learns from experience E, in a class of tasks T, given a measure of performance P, if their performance in the tasks contained in T, measured by P, improve with experience E."* With this definition in mind, machine learning provides a significant portion of the technical foundation for the development and improvement of learning systems using algorithms that pair experience and data.

Three learning paradigms are used in this domain: supervised learning, where learning is based on a set of examples with a vector of inputs and a vector of desired outputs; unsupervised learning, where learning is done by looking for patterns in the input data; and reinforcement learning, where rewards or punishments are only given through the model's responses in specific circumstances.

The neural network-based subset of machine learning, which enables a machine to teach itself to execute a task and attempts to identify underlying relationships in data in a manner similar to how the human brain works, should also be mentioned and explained.

This section will address supervised learning, unsupervised learning, and neural networks by outlining their fundamental principles and frequent applications, as well as by exemplifying each of them and highlighting any drawbacks. Since reinforcement learning must be described and thoroughly analyzed in order to comprehend the goal of this work and the suggested solution, an entire section will be devoted to it.

2.1.1 *Supervised Learning*

A model can be derived from a set of training samples with known inputs and outputs using supervised learning methods.

There are two main stages to it. The first step is training the model using inputs and outputs that are known, during which it typically tries to estimate the value of some parameters in order to minimize an objective function. The second stage is the prediction stage, where the model is applied to data in order to forecast the value of the output characteristics while only the values of the input attributes are known.

Calculating error measures over a given set of examples (preferably not used in the training of the model) allows one to assess the suitability of a model for a certain task. These error measurements vary based on the type of the problem (classification or regression).

Predictions regarding the category of a particular observation form the basis of a classification problem. Here, an effort is made to estimate a classifier that categorizes unseen data qualitatively based on the input data (including observations with already defined classifications). For instance, a classifier can identify if unobserved patient data is indicative of a disease or not.

A regression problem is similar to a classification problem in the sense it uses observed input data to predict a response. The main distinction is that, in this case, the goal is to estimate a numerical value rather than categorize an observation. An example is to estimate a model that uses an unobserved individual's age and years of schooling to predict the salary.

Some of the most well-known and often employed supervised learning techniques include linear regression, logistic regression, k-nearest neighbor, decision trees and random forests.

Linear regression [13] is performed in order to determine the relationship between a dependent variable and one or more independent variables and it aims to plot the best-fitting straight line, calculated using the least squares method [14]. When a dependent variable is categorical (has binary outputs like true or false or yes or no) logistic regression [15] is used rather than linear regression (used when the dependent variable is continuous). Both regression techniques seek to comprehend the connections between the data inputs but logistic regression is mainly used to solve binary classification problems, such as spam identification (spam or not spam).

K-nearest neighbor [16] is often utilized for recommendation engines and image recognition, and can be applied for both classification and regression problems. It comprises a non-parametric algorithm that classifies data points based on their proximity and association to other available data. This algorithm assumes that similar data points can be found near each other and as a result, it seeks to calculate the distance between the data points. Then it assigns a category based on the most frequent category or average.

Decision tree [17] is a method that predicts response values by learning decision rules drawn from observations on an item (represented in branches) to inferences about the item's target value (represented in the leaves). They can be applied in both classification and regression scenarios.

Random forest [18] is also a flexible supervised machine learning algorithm used in both regression and classification contexts. The forest refers to a group of uncorrelated decision trees that are combined to lower variance and provide more precise data predictions.

Disadvantages

Overfitting is one of the main issues with supervised learning. The model loses its ability to generalize when it learns from examples too well. For instance, the error in the training examples decreases but the model fails to fit additional data or predict future observations. This can be prevented by adding more data, employing cross-validation (resampling method that uses different portions of the data to test and train a model on different iterations), avoiding overly complex models in comparison to the available data, and avoiding excessively prolonging the learning process. Still, in reality, it is exceedingly challenging to find a solution to this issue.

In contrast to unsupervised learning, this technique is unable to extract unknown information from training data because it is unable to classify or group the data by independently identifying its features and furthermore,

unlike reinforcement learning where the learning agent itself generates data by interacting with the environment, supervised learning requires huge amounts of data (difficult to obtain) to train a system.

Ultimately, for more complex tasks supervised learning is relatively constrained and difficult to apply because it only learns from the input data and the unwanted data reduces efficiency and increases computation time.

2.1.2 *Unsupervised Learning*

The goal of unsupervised learning is to use a large set of variables to discover a smaller set of uncorrelated variables that can explain the majority of the variation in the data. Unsupervised learning is a very effective paradigm for discovering patterns in data since there is no target variable, making each record independent and removing the need for labels to provide orientation. In real life, unsupervised learning can be used to split the customers of a store into groups based on certain patterns, so that each group of customers is different while customers within a group share common properties.

For this type of learning, the most popular method is clustering [19] which aims to group observations into clusters where these observations exhibit similarities within their cluster and differences in relation to the other formed clusters. There is no labeling of clusters, in contrast to supervised learning classification methods, hence there is no wrong or right clustering. An example of this method is grouping photos of similar animals into clusters, without having prior knowledge of which animal is being presented.

Disadvantages

Overfitting is also a common problem in unsupervised learning. It is more often discussed as automatic determination of optimal cluster number and unlike supervised learning, the solution of using cross-validation is not applicable in this setting. This issue arises from the fact that many datasets lack distinct clusters, making it likely for two persons to offer different answers when asked to visually indicate the number of clusters by looking at a chart. Also, sometimes clusters overlap with each other, and large clusters contain sub-clusters, which makes finding an optimal cluster number difficult.

Additionally, it is often more challenging than supervised learning because it may need human assistance to comprehend the patterns and correlate them with the domain knowledge. And furthermore, because there are no labels or output measurements to demonstrate its effectiveness, it is not necessarily assured that the result of unsupervised learning would be beneficial.

2.1.3 *Neural Networks*

The brain is a tremendously complex system that can arrange its neurons to carry out intricate tasks. Since a neuron is five to six times slower than a logic gate, the brain overcomes this slowness by using a parallel structure.

As a machine learning method that simulates the functioning of the human brain, neural networks are parallel processors consisting of simple processing units (neurons). Knowledge is stored in the connections between

neurons and is learned from the outside world (data) through a training algorithm, which adjusts the connection weights.

This technique's success is primarily attributable to the ability to perform difficult tasks faster than other machine learning methods. It is helpful for a wide range of real-life situations because it can adapt its topology in response to changes in the environment, ignore noise and irrelevant qualities, handle incomplete information efficiently, and does not require explicit domain expertise (allowing it to be used as a black box).

As depicted in [6, Fig. 1], an Artificial Neural Network (ANN), also known as a feed forward neural network since inputs are exclusively processed in the forward direction, has at least three layers: input, hidden, and output. Inputs are received by the input layer, processed by the hidden layer, and subsequently the output layer produces the result.

An ANN can be used to solve problems related to tabular data or text data, as it has the capacity to learn connection weights that map any input to the output with activation functions that introduce nonlinear properties to the network. For this reason, this type of network is often referred to as a universal function approximator.

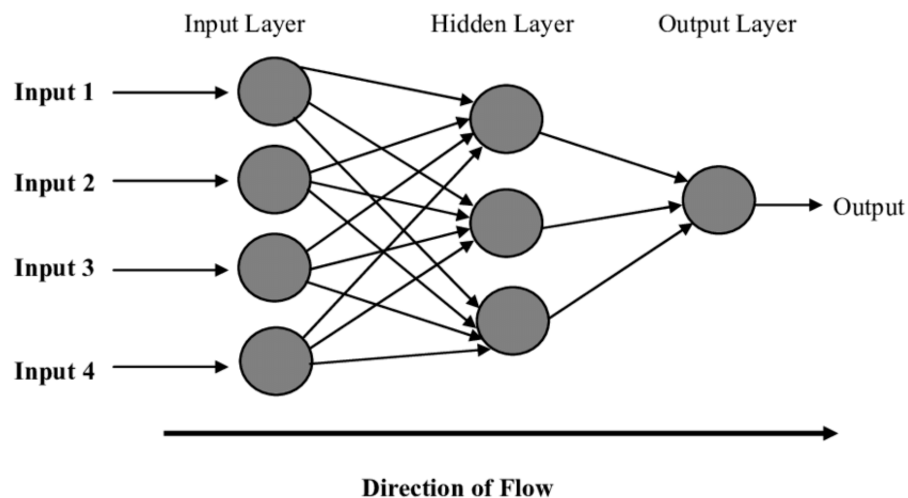


Figure 1: ANN structure from [6].

A single neural network differs from a deep learning algorithm or Deep Neural Network (DNN) [20] by the number of node layers (or depth). To create the last it is important to have an input layer, an output layer and more than just one hidden, middle layer (there are usually many hidden layers, allowing for more complex patterns to be learned). If there is just one hidden layer it is called a shallow network and it is not a deep learning algorithm.

Deep learning can be combined with supervised, unsupervised or reinforcement learning and it can be thought of as a subset of machine learning that's based on neural networks and that nowadays is the absolute cutting edge of artificial intelligence (a science devoted to making machines think and act like humans).

Disadvantages

The inability of neural networks to explain how or why a specific solution was formed (its black box nature) diminishes their credibility. Additionally, there is no set formula for figuring out how neural networks should be structured, therefore finding the right network topology is typically a matter of trial and error and expertise.

In general, neural networks require more processing power than conventional algorithms, and the amount of computational power required for a neural network is mostly influenced by the volume of the input data, but also by the depth and complexity of the network. For example, a neural network with one layer and fifty neurons will be much faster than a random forest with a thousand trees. By comparison, a neural network with fifty layers will be much slower than a random forest with only ten trees.

They also normally require significantly more data than traditional machine learning techniques and although there are certain instances where neural networks operate effectively with less data, most of the time they do not. Therefore, many machine learning problems can be more efficiently resolved with less data using other algorithms instead of neural networks.

2.2 REINFORCEMENT LEARNING

Reinforcement learning does not require labels or training data and can adapt to various circumstances because it learns from mistakes. In [7, Fig. 2] it is shown the basic idea and elements involved in a reinforcement learning method.

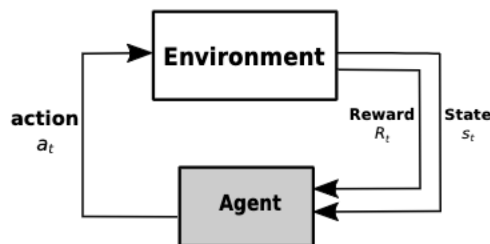


Figure 2: Reinforcement learning simple procedure from [7].

Reinforcement learning consists of one or more agents that take actions to interact with an environment. After each interaction, the agent receives a response (reward and state) from the environment. Its objective is to maximize the cumulative rewards received over an infinite number of interactions. To entirely understand this concept, its basic terminologies are better clarified next.

- **Agent**: An entity that tries to learn the best way to perform a specific task.
- **Environment**: Refers to the outside world in which the agent operates.
- **Action**: What the agent does at each step.
- **State**: Describes the current situation. After performing an action, the agent can move to different states in the environment.
- **Reward**: Feedback that is delivered to the agent in response to the agent's action. Positive rewards are given to agents whose actions are successful and have the potential to result in success, and vice versa.

With this in mind, most of reinforcement learning problems can be considered and solved as Markov decision processes [21] with a set of states, a set of actions, the probability of state *s* to state *s'* at time *t* under action *a* and the immediate reward after transaction from *s* to *s'* with action *a*.

Exploration versus Exploitation Dilemma

In order to maximize the cumulative rewards, this paradigm must strike a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

An agent can profit in the long run by expanding its current understanding of each action through exploration. On the other hand, with exploitation, the agent chooses the greedy action to get the most reward but being greedy may actually lead to suboptimal behavior.

Epsilon-greedy [22] is a straightforward technique to balance exploration and exploitation by either making the agent select a random action with probability $epsilon$ (helping the algorithm not get stuck in a local optimum) or by exploiting the best known action with probability $1 - epsilon$.

If epsilon is set to 0, it never explores but always exploits the knowledge already known. On the contrary, having the epsilon set to 1 forces the algorithm to always take random actions and never use past knowledge. The idea is having a big epsilon at the beginning of the training of the function and then reducing it progressively as the agent becomes more confident at estimating values.

Functions

The agent takes actions based on policy functions and value functions. Figure 3 introduces a brief summary on the difference between them. A policy function defines how an agent acts from a specific state. For a deterministic policy, it is the action taken at a specific state. For a stochastic policy, it is the probability of taking an action a given the state s. The value function measures how correct it is for an agent to be in a given state (state value function), or how correct it is for the agent to perform a given action in a given state (state-action value function). Value functions are always conditional on some policy.

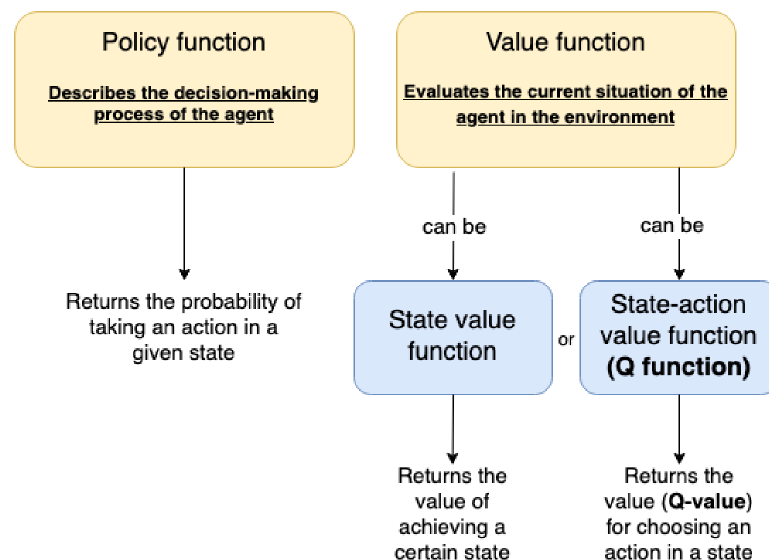


Figure 3: Reinforcement learning functions.

In the literature:

- The policy function is denoted by π and the optimal policy function is denoted by π^*
- The state value function is denoted by $V^{\pi}(s)$ and the optimal state value function is denoted by $V^*(s)$.
- The state-action value function(Q function) is denoted by $Q^{\pi}(s, a)$ and the optimal Q function is denoted by $Q^*(s, a)$
- $V^{\pi}(s)$ expresses the expected value of following policy π forever when the agent starts following it from state s.
- $Q^{\pi}(s, a)$ expresses the expected value (denoted Q-value) of first taking action a from state s and then following policy π.

Agents

In reinforcement learning, the agent tries to determine the most effective approach to carry out a task. As shown in Figure 4, they can be divided into two categories and a combined approach for better understanding of the scope.

Policy optimization agents can learn the parameters of a policy function directly. In this case, agents can directly tell which action to take. The value optimization category includes types of agents where the policy is learned indirectly by estimating a value function. A combined approach to value optimization and policy optimization is the actor-critic category. Two networks are used: actor and critic. The actor chooses what action to do, and the critic advises the actor on the actions quality and how to improve it.

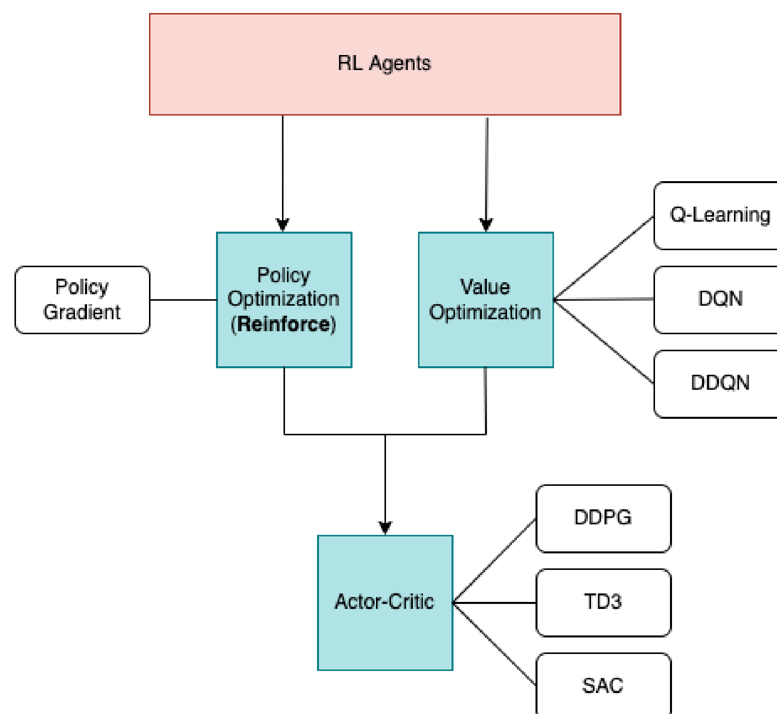


Figure 4: Reinforcement learning agents.

2.2.1 *Policy Optimization: Policy Gradient*

Policy Gradient (PG) [23] can directly perform gradient descent or gradient ascent on the policy function. Gradient descent [24] is an iterative optimization algorithm for finding a minimum of a differentiable function (function whose derivative exists at each point in its domain). It takes repeated steps in the direction opposite to the function's gradient at the current point as shown in Figure 5. Gradient ascent [25] is the inverse of gradient descent and is used to maximize the differentiable function. For example, gradient descent can be used when there is an error function to minimize and gradient ascent can be used when there is a score function to maximize.

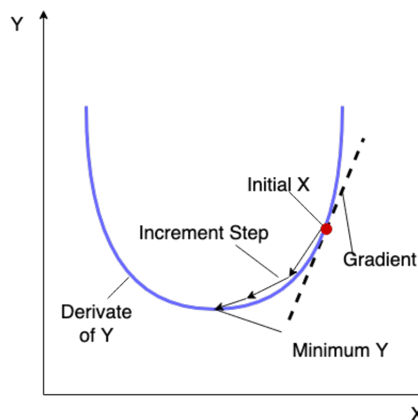


Figure 5: Gradient descent.

With PG there is a smooth update of the policy at each iteration. It follows the gradient to find the best parameters, so if there is an attempt to optimize a score function, for example, it will be guaranteed to converge on a local maximum (worst case) or global maximum (best case).

The advantage of the PG approach is that it is more likely than value optimization methods to converge on an optimal solution, making it more generally applicable. The trade-off is that this technique typically requires more training samples, performs inconsistently, and exhibits large performance volatility.

Policy gradients are also more effective when using continuous actions because the parameters are changed directly and because they can learn a stochastic policy, while value functions cannot. A stochastic policy, as mentioned previously, allows the agent to explore the state space without always taking the same action because it outputs a probability distribution over actions. As a consequence, it handles the exploration versus exploitation dilemma better for continuous actions problems.

2.2.2 *Value Optimization: Q-Learning, DQN and DDQN*

Q-Learning [26] discovers an optimum policy given unlimited exploration time and a partly-random policy, for any finite Markov decision process, in the sense of maximizing the anticipated value of the total reward beginning from the current state and taking an action (Q function).

The table (Q-table) is a basic data structure for keeping track of states, actions, and predicted rewards. The Q-table is initialized to all zeros at the start indicating that the agent knows nothing about the world. As the

agent takes various actions at various states via trial and error, the agent learns the anticipated reward for each state-action combination and updates the Q-table with a new Q-value, as shown in Figure 6.

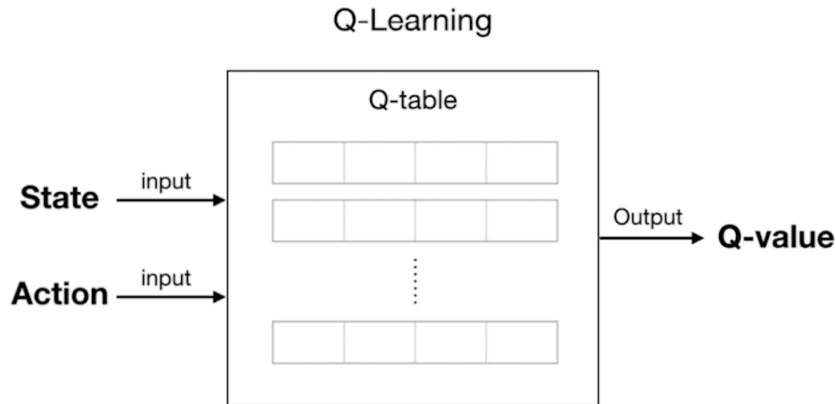


Figure 6: Q-Learning from [8].

The Q-table is updated using the Bellman equation [27] after each iteration. To summarize this equation, the agent updates the current perceived value with the estimated optimal future reward, assuming that the agent does the best currently available action. In a real-world implementation, the agent will go through all of the actions for a given state and select the state-action pair with the highest corresponding Q-value.

With the epsilon-greedy algorithm, every step the agent takes at the start of the algorithm will be random, which is beneficial for helping the agent learn about the environment it is in. As the agent takes more and more steps, the value of epsilon decreases and the agent begins to try existing known good actions more and more. Near the end of the training process, the agent will be exploring much less and exploiting much more.

Deep Q-Learning (DQN) substitutes the traditional Q-table with a function approximator (neural network) as shown in [8, Fig. 7]. The neural network maps input states to (Q-value, action) pairs rather than mapping an action-state pair to a Q-value. These inputs are also designated action-values.

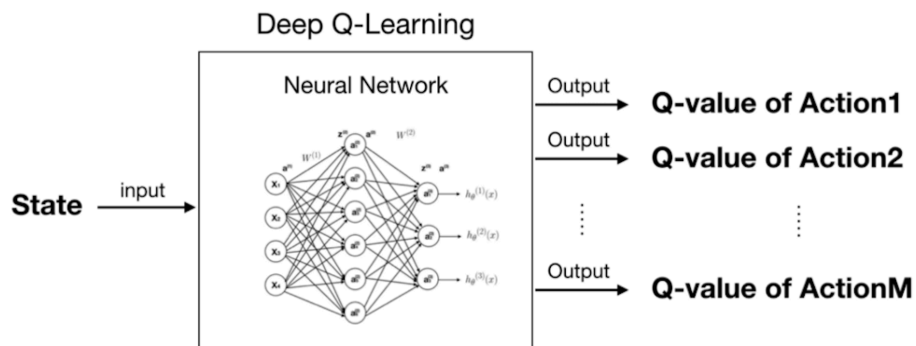


Figure 7: Deep Q-Learning from [8].

DQN updates many action-values at each step. This is a downside, as it can affect the action-values for the very next state instead of guaranteeing them to be stable as they are in Q-Learning. This happens all the time with DQN when using a standard deep neural network and the effect is referred to as catastrophic forgetting [28].

To counter this effect, DQN employs two neural networks in the learning process. The design of these networks is the same, but the weights are different. It uses a stable target network as a error measure (loss) for the Bellman equation and the weights from the main network are replicated to that target network only every N steps, allowing for the error measure not to vary at each step and in consequence not affecting the action-values for the next state. By giving the network more time to consider many actions that have taken place recently instead of updating all the time, it hopefully finds a more robust solution before it starts using the network to provide actions.

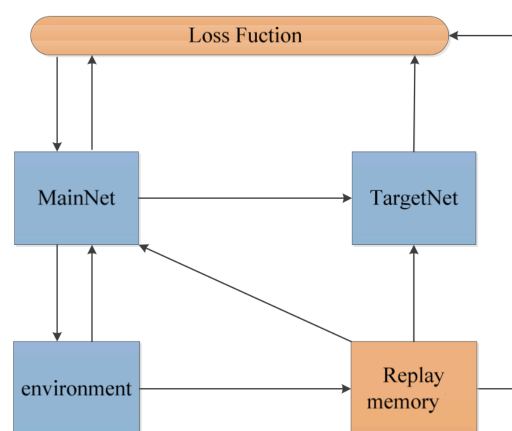


Figure 8: The principle of DQN algorithm from [9].

Also, when a nonlinear function approximator, such as a neural network, is employed to represent the Q function, reinforcement learning becomes unstable or divergent. The correlations in the sequence of observations, the fact that small updates to the Q function may drastically modify the agent's policy and the data distribution, and the correlations between the Q function and the target values all contribute to this instability.

To fight this problem, DQN agents use experience replay to learn in small batches about their environment and update the main and target networks. Experience replay [29] is a biologically inspired mechanism that uses a random sample of prior actions instead of the most recent action to proceed.

This removes correlations in the observation sequence and smooths changes in the data distribution. Iterative updates adjust the Q function towards target values that are only periodically updated, further reducing correlations with the target.

In Figure 8, it is described how these networks and the replay buffer are interconnected in the DQN agent.

Double Deep Q-Learning (DDQN) [30] solves one of the DQN algorithm's flaws: overestimating the true rewards (the Q-values predict that the agent will get a higher return than it would in reality). The straightforward solution consists in separating action selection from action assessment. The main neural network determines which of the potential next actions is the best, and the target neural network assesses this action to determine its Q-value. This simple technique has been demonstrated to prevent overestimation, resulting in improved policy outcomes.

2.2.3 *Actor-Critic: DDPG, TD3 and SAC*

Deep Deterministic Policy Gradient (DDPG) [5] interleaves learning an approximation to the optimal state-action value function with learning an approximation to the optimal action (policy function), and it does so in a way which is specifically adapted for environments with continuous action spaces.

The policy function structure is known as the actor, and the value function structure is referred to as the critic. The actor produces an action given the current state of the environment, and the critic produces a TD (Temporal-Difference) error signal given the state and resultant reward, as shown in Figure 9. If the critic is estimating the Q function, it will also need the output of the actor. The output of the critic drives learning in both the actor and the critic.

This method is closely connected to Q-Learning, and is motivated the same way: *if* it is known the optimal state-action value function (Q function), then in any given state, the optimal action can be found by finding the maximum Q-value. When there are a finite number of discrete actions, finding the maximum poses no problem, because it can just compute the Q-values for each action separately and directly compare them (this also immediately gives as result the action which maximizes the Q-value). But when the action space is continuous, the space can not be exhaustively evaluated, and solving the optimization problem is highly non-trivial because using a normal optimization algorithm would make calculating the maximum a painfully expensive subroutine.

When the action space is continuous, the Q-value function is presumed to be differentiable with respect to the action argument. This allows the algorithm to set up an efficient, gradient-based learning rule for a policy which exploits that fact. Then, instead of running an expensive optimization subroutine each time there is a wish to find the maximum, that maximum can be approximated with the policy.

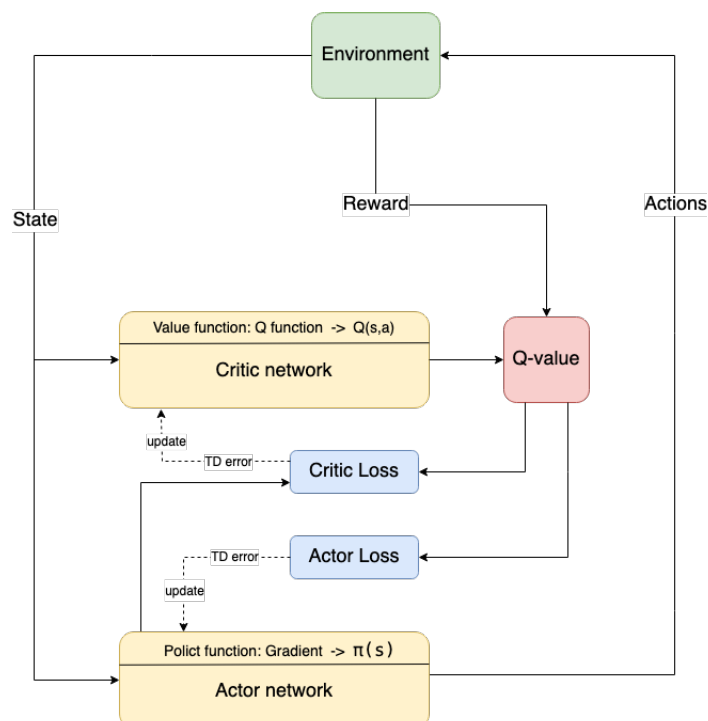


Figure 9: DDPG.

The policy learning in DDPG is fairly simple. Because the action space is continuous, and there is an assumption that the Q function is differentiable with respect to action, gradient ascent can be performed (with respect to policy parameters only) to solve the problem.

DDPG is based on DQN methods so it uses the same tricks as them, like replay buffers, target networks and the Bellman equation to update the weights of the neural networks.

Twin Delayed DDPG (TD3) agent is an actor-critic reinforcement learning method that searches for an optimal policy that maximizes the expected cumulative long-term reward.

While DDPG can achieve great performance, it may frequently be brittle with respect to hyperparameters. A common failure mode for DDPG is that the learned Q function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q function. TD3 is a method that addresses this issue by introducing three critical tricks:

- **Clipped Double Q-Learning**: TD3 learns two Q functions instead of one (hence twin), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions. Both Q functions use a single target, calculated using whichever of the two Q functions gives a smaller target value and then both are learned by regressing to this target. Using the smaller Q-value for the target, and regressing towards that, helps fend off overestimation in the Q function.

- **Delayed Policy updates**: TD3 updates the policy less frequently than the Q function. The paper recommends one policy update for every two Q function updates. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

- **Target Policy Smoothing**: It essentially serves as a regulator for the algorithm. It addresses a particular failure mode that can happen in DDPG: if the Q function approximator develops an incorrect sharp peak for some actions, the policy will quickly exploit that peak and then have brittle or incorrect behavior. This can be averted by smoothing out the Q function over similar actions, which target policy smoothing is designed to do.

Together, these three tricks result in substantially improved performance over baseline DDPG. To make TD3 policies explore better, noise should be added to their actions at training time, but to facilitate getting higher-quality training data, the scale of the noise over the course of training may be reduced.

Soft Actor-Critic (SAC) [31] is a method that optimizes a stochastic policy, forming a bridge between stochastic policy optimization and DDPG-style methods. It is not a direct successor to TD3 (having been published roughly concurrently), but it incorporates the Clipped Double Q-Learning trick and it also winds up benefiting from something similar to Target Policy Smoothing without having to implement it.

A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy. The term entropy has a rather esoteric definition and many interpretations depending on the application but one can think of entropy as how unpredictable a random variable is. If a random variable always takes a single value then it has zero entropy because its not unpredictable at all. If a random variable can be any real number with equal probability then it has very high entropy as it is very unpredictable. A high entropy is necessary in the policy to explicitly encourage exploration, to encourage the policy to assign equal

probabilities to actions that have same or nearly equal Q-values, and also to ensure that it does not collapse into repeatedly selecting a particular action that could exploit some inconsistency in the approximated Q function and prematurely converging to a poor local optimum.

The modern version for SAC makes use of three networks: two Q functions and a policy function π. The Q functions are learned in a similar way to TD3, but with a few key differences. Like in TD3, both Q functions are learned by minimizing the error in the Bellman equation (by regressing to a single shared target).

Unlike in TD3, the target also includes a term that comes from SACs use of entropy regularization and the next state actions used in the target come from the current policy instead of a target policy.

2.3 MACHINE LEARNING TUNING SYSTEMS

Tuning optimization of a system is an NP-hard problem [2]. This topic focuses primarily on Database Management systems (DBMS) and how there are hundreds of knobs to tune and how expensive it is to find high-quality configurations for the same systems. Among these are OtterTune, CDBTune or QTune. Even though the aim of this work is not achieving an optimization of DBMS systems, these are the papers that are more related to the problem of creating an autonomous optimization for a transactional management system.

2.3.1 *OtterTune*

OtterTune [1] is an external knob configuration tuning service that keeps a repository of information gathered from earlier tuning sessions and uses this information to create models with a variety of metrics on how the DBMS reacts to various knob configurations. These models are used to direct experimentation and provide ideal settings for a new application. In a feedback loop, each recommendation gives the service new data, allowing it to hone its models and improve their accuracy. It chooses the most influential knobs, maps unknown database workloads to existing workloads, and suggests knob settings using a combination of supervised and unsupervised learning techniques.

In terms of the analysis and built of the models, as shown in [1, Fig. 10], the process starts in OtterTune's repository and it's data is first passed into the workload characterization component, which identifies the most distinctive DBMS metrics by first gathering all the statistics at the beginning of each observation and then pruning redundant metrics to narrow the search space of the machine learning algorithms. To do this, they use a dimensionality reduction technique (factor analysis [32]), which condenses a set of real-valued variables with arbitrary correlations into a smaller set of factors that accurately reflect the correlation pattern of the original variables. Then, they cluster this lower dimensional data into useful groupings using a second methodology, an unsupervised learning approach called k-means [33]. Dimensionality reduction techniques are often used as a preprocessing step before clustering methods are applied because they reduce the amount of noise in the data [1]. After eliminating the redundant metrics, OtterTune uses a LASSO [34] algorithm to choose the most impactful and significant knobs.

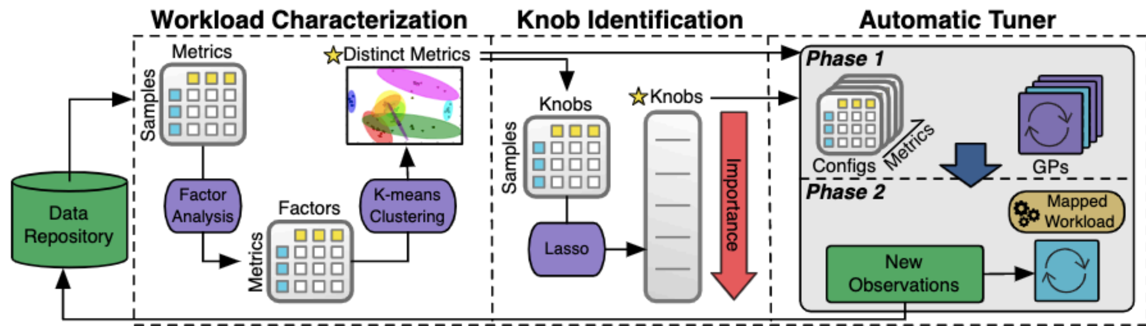


Figure 10: OtterTune machine learning pipeline from the original paper [1].

At this point OtterTune's repository contains a collection of non-redundant metrics, a set of the most impactful configuration, and the data from earlier tuning sessions. OtterTune repeatedly analyzes the data it has collected so far in the session and then recommends the next configuration to try.

In Phase 1, the system determines which workload from a prior tuning session is the most representative of the target workload by comparing the metrics to see which ones respond similarly to various knob settings.

OtterTune uses Gaussian Process (GP) [35] regression in Phase 2 to suggest configurations that it believes will improve the target metric. It starts by training the model and updating it by adding in the metrics from the target workload that it has observed so far. OtterTune then employs an exploration versus exploitation strategy in the GP to try to find a configuration that is superior to the best configuration it has seen thus far for that session. It also uses gradient descent to identify the local optimums for the case of exploitation and once it selects the next configuration, it returns that configuration to the client along with the expected improvement from running the selected configuration.

Advantages and Problems

The configurations produced by this system achieve a lower latency in comparison to default or generated by other tuning advisors settings.

The biggest issue with this method is that it relies on a large number of high-quality training examples from training datasets that are difficult to find.

Additionally, it uses a simple regression model (GP) for recommendation which is still too simple compared to a neural network model and cannot explore new knowledge to refine itself because it does not search high-dimensional configuration space when the workloads or hardware configurations differ from training conditions.

As a result, especially in a cloud context, the performance of the recommended configuration is very limited when the external environment changes, and the lack of pertinent data in the training dataset will directly result in a poor recommendation to OtterTune [2].

Furthermore, OtterTune filters twice (factor analysis and k-means) to obtain the most crucial metrics, and then again (LASSO) to gain the most important knobs. This filtering process can result in information loss, because even though the filtered data might be less important than the chosen ones, it can still have an impact on the database's performance.

2.3.2 *CDBTune*

CDBTune [2] is an end-to-end automatic configuration system that can recommend superior knob settings in the complex cloud environment by using a try-and-error manner to learn the best settings with limited samples and by finding the optimal configurations in high-dimensional space [2].

It uses the DDPG algorithm for reinforcement learning mentioned before which can process high-dimensional states and generate continuous actions that can straightforwardly predict the values for all tunable knobs.

As depicted in [2, Fig. 11], when the client initiates a tuning request or the database administrator initiates a training request via the controller, the workload generator performs stress testing on the CDB's instances that still need to be tuned by simulating workloads or replaying the user's workloads. At the same time, the metrics collector collects and processes related metrics. The memory pool will store the processed data and then feed it to the deep reinforcement learning network.

When the model finally outputs the suggested configurations, it generates the necessary execution setting parameter commands and sends the controller a request to adjust the configurations. The controller deploys the aforementioned configurations on CDBTune's instances after obtaining the database administrators or users license. The model is also updated by CDBTune using the tuning or training request as training data.

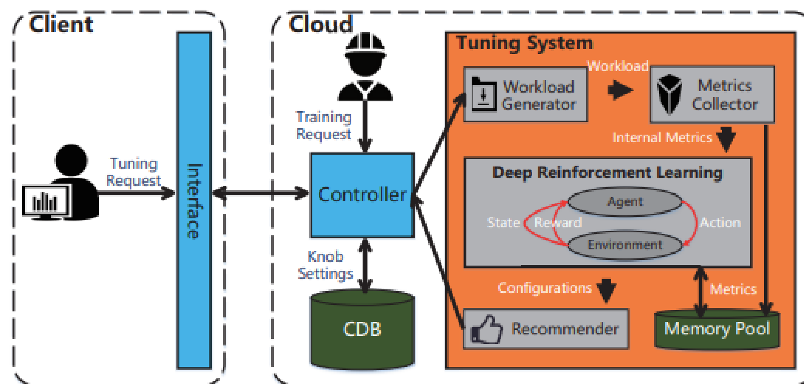


Figure 11: CDBTune system architecture from the original paper [2].

Advantages and Problems

Compared to OtterTune, the DDPG algorithm can give a better configuration recommendation in high-dimensional continuous space than the simple regression model OtterTune used. In the absence of high-quality empirical data, gaining experience through trial-and-error methods makes it easier to collect data and helps the model generate a variety of samples and learn in an optimizing direction using deep reinforcement learning, which can effectively use small samples to produce significant results.

Although CDBTune might not find the global optimum, reinforcement learning uses the exploration and exploitation dilemma to not only maximize the model's potential but also to investigate additional optimal configurations that the database administrator had never considered, thereby lowering the risk of becoming trapped in a local optimum.

One problem with this system is that it requires to run a SQL query workload multiple times in the database to initially train the model, which is rather time consuming.

2.3.3 *QTune*

A query-aware database tuning system with a deep reinforcement learning model is proposed by QTune [3] to efficiently and effectively tune a database's configurations.

It employs concepts that are extremely similar to CDBTune's, but with a crucial distinction. It extracts features from queries into vectors in order to distinguish them and pinpoint crucial components in the query that may indicate which configuration would be more appropriate. It does this by using information from the databases engine query optimizer.

In [3, Fig. 12] a variation of DDPG is proposed, the Double-State Deep Deterministic Policy Gradient (DS-DDPG) method.

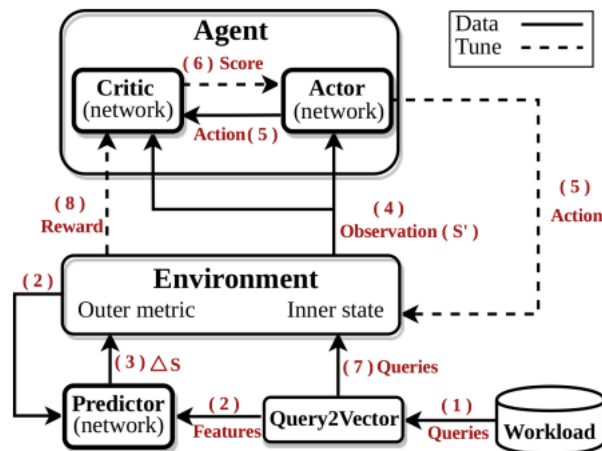


Figure 12: DS-DDPG method from the original paper [3].

The environment holds the database information, which comprises the inner state and the outer metrics. The use of these two metrics is what gives this new version its name (double state variation). The inner state records the database configuration (knob settings) which can be tuned, and the outer metrics record the state statistics (database key performance indicators), which represent database status and cannot be tuned.

Each query is converted into a vector via Query2Vector. In order to create a vector, it first evaluates the SQL query and extracts the query plan and the estimated cost of each query from the database engine. A deep neural network serves as the predictor and forecasts changes in outer metrics.Based on the observation created by the environment, the agent is used to tune the inner state.

The agent contains two modules, actor and critic, which are two independent neural networks based on the DDPG algorithm.

The actor takes the observation as input, and outputs an action (a vector of tuned knob configurations). The query workload is executed by the environment, and a reward is calculated based on the performance. A score (Q-value), which indicates whether the action tuning was successful, is produced by the critic using the observa-

tion and the action as input. Based on the reward value, the critic updates the weights of its neural network and based on the Q-value, the actor updates the weights of its neural network.

As a result, the actor creates a tuning action, the environment implements it, and generates a reward value based on the performance change of the new configuration. A positive reward will be given if the performance change is positive; otherwise, a negative reward will be given.

Advantages and Problems

QTune offers an efficient strategy for solving optimal problems with continuous action space by concurrently learning the Q-value function and the action policy.

Despite the fact that QTune has an advantage over CDBTune thanks to the usage of query information, getting data from the engine's query optimizer requires a thorough understanding of the engine's internal mechanisms. This means that QTune will not integrate easily with different systems.

2.3.4 *Multi-Model Tuning Algorithm*

Multi-Model Tuning Algorithm [4] also uses DDPG to learn a policy function for the tuning agent, but instead of one model, uses multiple DDPG models for workloads that change in an online tuning environment.

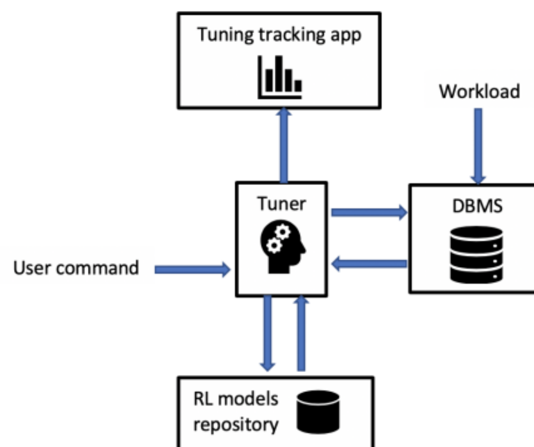


Figure 13: Multi-Model database online tuning system [4].

Since workloads can vary at any time, tuning online is a difficult task. This issue is resolved by the multi-modeling approach illustrated in [4, Fig. 13] tracks the database state and performance. The agent employs a set of pre-trained models and dynamically produces new models to tune the knobs if required.

The main components of the system are the Tuner and the reinforcement learning models repository. The Tuner refers to the DDPG reinforcement learning version already explained before. The models repository includes both models developed during online tuning and models that have been pre-trained on workloads that are as similar to the workloads in the deployment environment as possible. The weights, replay buffer, and log file containing the knob settings that produced the greatest training database performance are all included with the models in the repository. Additionally, a workload representation vector that enables retrieval of the models that

are the most similar is provided with each model. The last component is a web application that uses the log files created during training to display in real-time the database performance measures and the values of the knobs being tweaked.

Advantages and Problems

According to preliminary results reported in the research, the multi-model strategy offers an advantage over the single-model approach, where only one model is continuously trained and needs to adjust to different workloads.

A possible drawback of the multi-model algorithm is that the ability to generate numerous new models can lead to a large number of competing models for every workload. However, in the paper it is observed that the pre-trained models were found to be superior to newly created models, and therefore, new models were not persisted. On this account, it is believed that eventually, the number of models in the repository will be comparable to or very close to the number of various workloads the agent is tuning.

2.4 TRANSACTION MANAGEMENT

The idea of a transaction consists in a sequence of operations with the following ACID properties [10]:

- **Atomicity**: All of the transactions' operations either succeed completely or not at all.
- **Consistency**: Transactions transfer the system from one legitimate state to another through the preservation of system constraints.
- **Isolation**: The execution of concurrent transactions preserves the semantics of the defined consistency criterion or isolation level.
- **Durability**: The effects of successful transactions are durable even in the presence of faults.

A transaction designates a portion of work on a database that must either be fully finished (commit) or have no impact (abort).

Large data is typically dispersed among thousands of servers and updated by a large number of clients, with node failures occurring frequently. Supporting transactions is essential in these circumstances to enable the system to handle partial changes brought on by faulty clients or if concurrent clients happen to update the same record in a database at the same time.

In situations where database transactions are required, software developers must address this issue at the application level because NoSQL systems often do not provide ACID transactions [36]. This constrains portability and creates the need to use system-specific code. pH1, which is employed in this study as a foreground for the suggested system, is one of several strategies that have been put out to address transaction management in NoSQL databases.

2.4.1 *pH1: A Transaction Middleware for NoSQL*

pH1 [10] is a middleware system that can offer a transactional interface with ACID guarantees to a common NoSQL database. While allowing operations to be bracketed in a transactional context using snapshot isolation as the isolation criterion, it still maintains the interface of the underlying NoSQL database. Snapshot isolation is based on optimistically executing concurrent transactions that are certified at commit time. If the isolation criteria is broken, i.e., if two concurrent transactions under snapshot isolation write to the same data item, they will clash. If the system recognizes the disagreement, then at least one transaction must be aborted.

Each transaction operates independently of the others, using a separate virtual snapshot of the database. In order to achieve this, pH1 implements a distributed multi-version cache of the database that is in constant sync with the permanent NoSQL database. This is referred to as the Non Persistent Version Store (NPVS), and it forms the basis of pH1 strategy.

In [10, Fig. 14], it is shown two pH1 instances. The middleware can scale by adding more pH1 instances to the system.

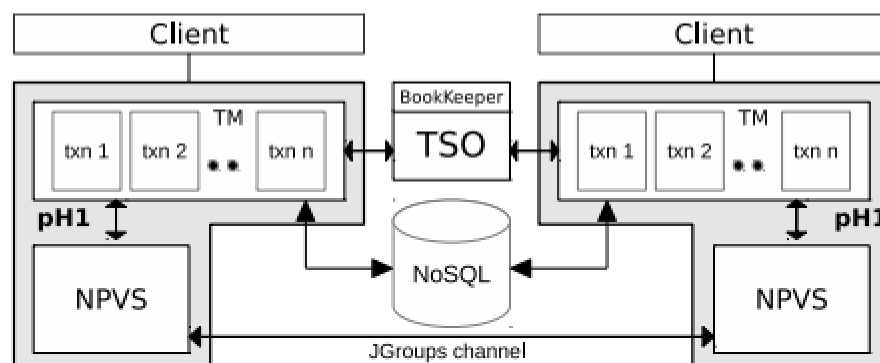


Figure 14: Overview of a configuration with two pH1 instances from the original paper [10].

Each instance has identical access to the Timestamp Oracle (TSO) module and the NoSQL database. The TSO is an external certifier and timestamp generator that Yahoo! [37] reused from the OMID project. The numerous NPVS instances communicate with one another using a group communication protocol.

One of pH1's core modules, the transaction manager, serves as a middleman between modules and exports and saves the transactional context of each active transaction in the system. The client will be given by the transaction manager a simple interface by the transaction manager that includes two actions for starting a new transaction and stopping an existing one. A private snapshot of the data items being modified in each transaction will be available (write, update and delete operations). This refers to the transactions write-set [10].

To support a wider variety of NoSQL databases, especially those that do not support multi-version tuples, the NPVS was developed in pH1 to handle and store tuple versions [10]. The transaction manager instance serves as the client for an NPVS node and the NPVS system specifies the kinds of messages that will be sent through the established channel in order to enable communication between nodes.

Cassandra

Cassandra is a distributed NoSQL database that is supported by pH1. It has a flexible approach to schema definition and horizontal scalability that is made possible by the distributed architecture it uses.

Facebook created Cassandra to handle a lot of write operations while attempting to achieve low latency read operations. It is renowned for having high availability (authorized users can access the information) and scalability (the ability of a system to handle an increasing amount of work by adding resources to the system) without sacrificing its performance.

Data in Cassandra is arranged in what are known as column families, which are structures that resemble tables in the relational model. The data objects are accessible through an API that allows simple put and get primitives.

Prior to version 2.0, Cassandra did not provide any ACID-compliant transactional guarantees of any kind. The idea of lightweight transactions is introduced in more recent Cassandra versions [38]. However, what actually is provided is a set of conditional read and put operations, which trade isolation and atomicity for high availability and quick write performance. With atomicity and isolation at the transaction level, Cassandra's API still differs from a fully transactional API.

2.4.2 *Related Work on Transaction Management Systems*

Beyond pH1, other notable transactional management systems are reviewed in this section, from the oldest to the most modern.

- Deuteronomy [39] distinguishes between the transactional and data components. To handle concurrency and assure consistency, it employs a locking mechanism. This locking is beneficial, however it has an impact on the transaction's performance.
- CloudTPS [40], as Deuteronomy, has a two-layer design that comprises LTM (Local Transaction Manager) and cloud storage. To maintain consistency in the event of failures, transactions are duplicated across LTMs.
- Granola [41] provides a solid consistency model while lowering transaction coordination overhead dramatically. It adds support for a new form of distributed transaction called independent distributed transactions, which may serialize with minimal locking cost and no aborts due to writing conflicts.
- Spanner [42] started out as a key-value API-based transactional NoSQL store with cross-shard transactions, external consistency, and transparent failover between data centers. Since then, it has developed into a widely distributed relational database management system that now powers almost all of Google's mission-critical services. A recent example is Google Spanner, which employs distributed locks along with a cutting-edge method called TrueTime that enables using local clocks for transaction ordering without the need for an expensive clock synchronization algorithm.
- Omid [43] implements a lock-free, centralized transactional scheme that can handle the high volume of transactions in large data stores. With no noticeable performance impact, it enables transactions for

applications running on top of data stores. Omid can be installed on top of any multi-version data store with a basic API resembling a key-value store.

- M-Key transaction model [44] uses a loosely coupled architecture to keep abstraction and transparency high while separating transaction processing from the underlying data storage. The suggested method also makes use of snapshot isolation to implement concurrency.

- The database engine in Azure Cosmos [45] support snapshot isolation and fully ACID compliant transactions. The database engine that is hosted by the replica of a partition performs all database operations within its confines. Write operations (updating one or more objects inside the logical partition) and read operations are both included in these operations.

- The surfing concurrence transaction model [46] offers a transaction model that enables NoSQL databases to perform transactions in a lock-free, multi-version concurrency control-free manner, enabling users to access data in an ACID manner. The model can effectively reduce the complexity of development for software systems with transaction demand.

2.5 DISCUSSION

After taking into consideration the advantages and disadvantages of the machine learning techniques, the paradigm chosen for the subject addressed in this dissertation is reinforcement learning (with neural networks) so that it can help the system learn continuously from experience. The other paradigms have various issues that will interfere in this project. For example, this task is overly complex (NP-hard problem) for supervised learning. And since there would be no labels or output measures to prove its efficacy, the outcome of unsupervised learning would not always be guaranteed to be advantageous.

The related work in the previous section should be taken in consideration as a basis perspective to resolve the problem at hand, but still, the critical judgment on how to adapt the literature papers to this project itself should always be present.

In [47] the authors extended the OtterTune tuning service to support several machine learning tuning algorithms in order to evaluate them and make a side-by-side comparison. Two of them were the Gaussian Process (GP) from OtterTune and Deep Deterministic Policy Gradient (DDPG) from CDBTune. The results showed the algorithms evaluated could generate knob configurations that outperformed those produced by a human expert by up to 45%; however, the performance was affected by the number of tuning knobs and the assistance of human experts in the knob selection.

Following a comparison of these algorithms, it was found that GP always converges quickly, but it can sometimes become stuck in a local minimum. Additionally, once it converges, it stops exploring, which means it cannot continue to improve after that point. In contrast, DDPG can converge and still carry out more exploration. It has the best performance because the continuous configuration space enables DDPG to identify the most appropriate settings for the relevant knobs while GP usually selects a sub-optimal version of the features to enable.

In this project, because these systems fit in the continuous space problems with few configurations, actor-critic methods will be used as guidelines, in particular the DDPG algorithm, but taking into consideration other variations like SAC and TD3.

In terms of the use case studied, pH1 is the chosen middleware to test our autonomous optimization system. Still, a lot of other transactional systems are being implemented in recent years and therefore, a solution should be created that can not only optimize a particular system of this type, but generalize to other systems that can use OPAL to tune their parameters.

The time it takes for pH1 to call the garbage collector is the focus of this dissertation. This was the parameter to be optimized chosen because it is one of the meaningful parameters that can be adjusted without having to restart the database or the middleware system, making it very important for online tests.

The garbage collector should impose a low added time overhead. With this in mind, our goal is to minimize that overhead by finding the optimal time for the garbage collector to take action. The variables that will be affected by this action are the write-set queues of transactions and the keys that exist in the cache of pH1. The write-set queues are the transaction queues that the middleware has internally. They serve to manage temporary items. The cleanup that OPAL induces tries to remove past items from memory, with an impact on performance.

The ultimate goal is to decide when to proactively clean these structures in order for the middleware to maintain a constant execution performance over time, capable of adapting to various workloads. It is important to keep in mind that these components are difficult to adjust (without being completely random).

3

OPAL

An approach to the optimization of transactional management systems that to the best of our knowledge has not been seen before in the literature is introduced in this chapter: OPAL - **OP**timization and **A**utonomous **L**earning for a transactional middleware.

3.1 ARCHITECTURE

OPAL is built with the optimization of a distributed system in mind, in particular, a transactional middleware system considering a black box perspective, leveraging from machine learning techniques, equipping the middleware with the ability to tune and adapt to any workload.

In order to create an architecture based on all the principals that were taken into account and previously mentioned (continuous learning, online tuning, etc.), two modules were created: the Wrapper module and the Learning module.

The Wrapper module is where it is obtained the metrics that will feed the reinforcement learning agent and where it is implemented the configurations coming from the provided action into the system that is being optimized. The Learning module is where the reinforcement learning agent and the entire process necessary for its learning are located. It was decided to make this separation since physically the components of one module are already separated with the components of the other module on a logic level and on a language level (java and python). This separation also helps understand how the OPAL pipeline works.

OPAL was created to be generic. So OPAL was not built with just a middleware system in mind. In fact, OPAL can be used not only with transactional middleware, but any system that is looking to optimize its configurations. This is done because in the Wrapper module, as shown in Figure 15, there is a Driver component that provides a bridge between the machine learning model (where the optimization happens) and the system being optimized. In this way, if a new system wants to use OPAL, it is only necessary to create a Driver for that system and implement the set of abstract operations defined by our solution, so that OPAL can support it.

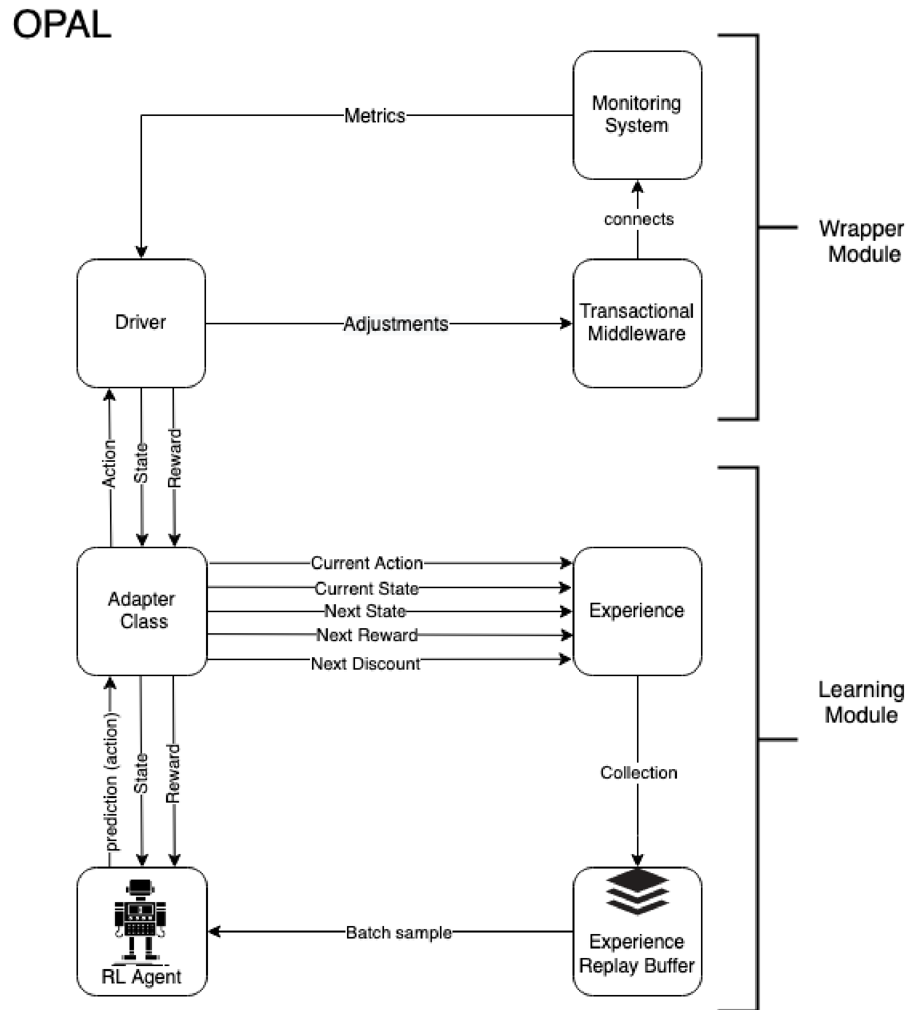


Figure 15: OPAL architecture.

In Figure 15 the Driver is connecting to a middleware system and to a monitoring system. It is connecting to a middleware system, because even though it can connect to any system, our focus is in a transactional middleware and it is connecting to a monitoring system because the Driver can get the necessary information from other sources (numerous ones) that are not the middleware system itself. There are transactional middleware systems that already provide a monitoring system in itself, or the underlying database already has a tool that can give enough information about the metrics to be used. Other times, the user of the system has to be the one to implement the monitoring system. The Driver is implemented with a flexibility to allow any of these cases.

Having multiple reinforcement learning agents that should be implemented, there is a need for the Learning module to also be extensible. The components of the Learning module in Figure 15 are the most general ones to provide an overview of the architecture.

The Learning module consists of an Adapter class working as an abstraction class that the agent can implement to get a better understanding of the environment. This class receives the Driver values via a socket connection and informs the agent about the state and the reward chosen for the system in that particular moment. At the same time it receives predictions from that agent itself.

The agent also receives a sample of experiences given from the replay buffer to train the neural networks. Those experiences are being created with all the information necessary from both the Adapter class and the reinforcement learning agent.

3.2 WRAPPER MODULE

In the Wrapper module there is a Driver class that is connected via a socket to a reinforcement learning agent, as shown in Figure 16. The Driver obtains the metrics from the middleware (or a monitoring system) and tunes the system.

The Driver was designed to assist OPAL to be as general as possible, contemplating an interface with mandatory methods to implement. Thus, it allows those who want to use OPAL in another middleware or system, the capability to change the project with the advantage that they only need to create a new Driver class that implements the Driver_Interface methods and thus can easily use the system in another environment.

These abstract methods for the user to change consist in methods that allow the user to choose which actions, observations and rewards (in terms of a reinforcement learning problem) are appropriate for the problem at hand. This means that if instead of wanting to reduce the latency, the user would like to increase the number of operations per second (in other words, change the reward of the algorithm), there would be no need to change code on the reinforcement learning agent implemented. Instead, in the Driver class, all the important configurations (number of actions, observations, etc.) can be changed.

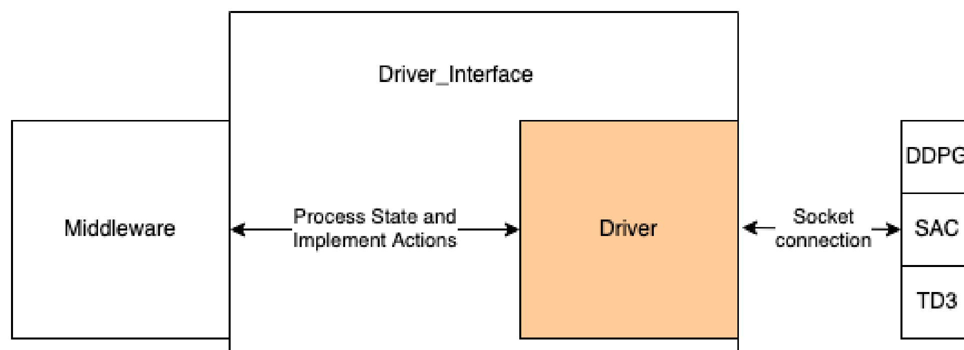


Figure 16: Wrapper interconnections.

Although it is possible to implement different scenarios and environments through the Driver, the following subsection (Monitoring System) focus on the choices made for the particular case of the middleware pH1.

3.2.1 *Monitoring System and Tuning Parameter*

In pH1 the monitoring system was tailor-made to obtain the most important metrics that will make the biggest difference.

To process the state, it was used the benchmark tool YCSB [48] to obtain the throughput and the latency (read or update depending on the workload), and other middleware statistics implemented in the tool. It was also

obtained the throughput (operations per second) from the Timestamp Oracle (TSO) logs, taking samples every 10 seconds to align with the time samples that are taken in the YCSB and corroborate them.

The action given to the middleware to implement is the time it takes for the garbage collector to take action.

Tuning this parameter was also done in a similar way as the implementation of the monitoring system, meaning that certain methods were overwritten in the Driver_Interface, allowing it to connect the action to be tuned with the method in the middleware that would allow it to happen. In our case, because the parameter was time, in the Driver method it was decided to wait the time the model would give us (the action) and then send a signal to activate the garbage collector.

3.3 LEARNING MODULE

In the Learning module it was built and adapted several reinforcement learning algorithms to our problem. An overall vision of how the Learning module is constituted is represented in Figure 17.

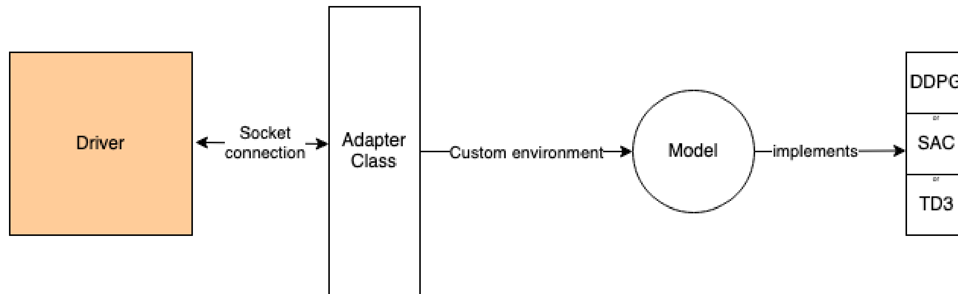


Figure 17: Learning module overview.

The Driver is connected to an Adapter class that can provide the model the environment specifications. In the opposite way, this Adapter class also provides the Driver with the right actions through the socket connection.

This architecture allows OPAL to be extensible and to implement different reinforcement learning agents for the same environment without any additional costs. The structure was built in a dynamic way, so adding another algorithm would be a very easy and accessible thing for the user to do, since it can easily connect to the Driver via a socket connection.

With the help of Tensorflow Agents (TF Agents) [49], different agents were implemented in OPAL, in particular, the actor-critic methods DDPG, SAC and TD3 to further evaluate and compare the three of them with each other. In Table 1, it shows the mapping from an actor-critic algorithm to our specific tuning problem.

Actor-critic method	Tuning
Environment	Middleware being tuned
State	Middleware statistics (e.g., latency)
Action	Choose the right time to call garbage collector
Reward	Middleware performance changes
Agent	The Actor-Critic networks
Actor	A neural network for making actions
Critic	A neural network for evaluating the Actor

Table 1: Mapping actor-critic algorithm to OPAL tuning.

Our problem consists in finding a good optimization for a system configuration that can have continuous values (e.g. time). With that in mind the actor-critic algorithms were chosen from all the other ones investigated because these are the most advanced algorithms dealing with actions in a continuous space. As mentioned before in an actor-critic method, the policy is referred to as the actor that proposes a set of possible actions given a state, and the estimated value function is referred to as the critic, which evaluates actions taken by the actor based on the given policy.

3.3.1 *Adapter Class / Custom Environment*

The environment is the surrounding or setting on which the agent performs actions. As explained in Table 17 the environment in the particular case for OPAL is the middleware/system being tuned. It was done all the important operations to implement our environment in the Driver, as shown before.

Despite this, in the Learning module, to connect with the TF Agents library, there is a need for an Adapter class that can be used in every model to guarantee that the environment that it is being used has the same properties for every model.

This Adapter class is usually used in TF Agents, to implement the environment for the model to use. However, in our case, the environment is implemented in the Driver, to allow for a better separation and abstraction. Therefore, the Adapter class is used as a way for the models to only handle our environment (already implemented) without needing to pass the socket information in a hard coded way.

For our situation, this Adapter class needed to be customized. TensorFlow calls this particular class an environment class because for them the environment is implemented in this class. For us it is not the same, as already explained, there is no need to implement any environment logic here, just pass values and information and tell the agent how it should handle our environment. Still it is preferred to refer to this class as our Adapter class or our custom environment class, since it is customized to our needs.

Environment classes can be easily built with Python or TensorFlow libraries. Python environment classes are simple to implement but TensorFlow environment classes are more efficient and allow natural parallelization. In this project, it is used use a Python environment class and then one of the TF-Agents wrappers to convert it to a TensorFlow environment class.

Creating a custom environment class constitutes of writing primarily four methods:

- **Action Spec:** Describes the specifications of the action expected by step. The specifications consist (in the case of OPAL) on the size of the array with the actions and on the minimum and maximum value that these variables can take.

- **Observation Spec:** Defines the specifications of observations provided by the environment. The specifications consist (in the case of OPAL) on the size of the array with the observations and on the minimum and maximum value that these variables can take.

- **Reset:** Returns the current situation after resetting the environment.

- **Step:** Applies the action and returns the new situation.

In the custom environment class, the functions only have to be set thanks to the TF Agents library and the Driver class by calling these values from the socket connection.

It is in the step method that most of the logic is coded. It receives the action chosen by the network as an argument, and must apply that action to the environment (for the next step) as well as retrieve the new state and calculate the reward associated with the undertaken action.

The step method returns the following information about the next step:

- **Observation:**The part of the environment state that the agent can observe to choose its actions at the next step.
- **Reward:** The agent is learning to maximize the sum of these rewards across multiple steps.
- **Discount:** Float representing how much to weigh the reward in terms of importance at the next step relative to the reward at the current step.

While writing the step method, it is important to see if the episode has ended. If it has ended, there is a need to call the reset method and give the agent the appropriate reward.

It is worth mentioning the difference between episode and step. An episode is an instance of a problem. If the problem ends, the episode ends. Step, on the other hand, is the time or some discrete value which increases monotonically in an episode. With each change in the state of the problem, the value of step increases until the problem ends.

In essence, this Adapter class created is a layer of abstraction between the model and the environment that helps to generalize the Learning module, so that any new model added to OPAL from TF Agents library can use an environment already defined. To do that there is only the need of calling the custom environment without needing to change anything else. Two different environment classes for our models were created: the train environment for training and the evaluation environment for evaluation of the models.

3.3.2 *Model Architecture*

The implemented architecture has two parts: collection part and training part, as it shows in Figure 18. The collection part collects experiences to feed the replay buffer. The training part uses an agent to train the networks.

Experiences consist of the current action, the current observation and the next step (next observation, next reward and next discount). They are recorded in logs, which are then aggregated over time and stored in the buffer.

The replay buffer is the buffer where experiences are stored, describing how the actions resulted in rewards. The batches of experiences are pulled from it to be fed into the agent so he can train the networks. When the batch is created in a random fashion from the replay buffer, the training experiences are identical and independent, helping in the training process using the gradient descent algorithm.

Once trained, the networks update the collect policy that is used to create experiences and deliver the predicted action to the environment after receiving the state and the reward from the previous action made.

In the agent, the actor and critic use two neural networks that generate the action probabilities and critic value respectively.

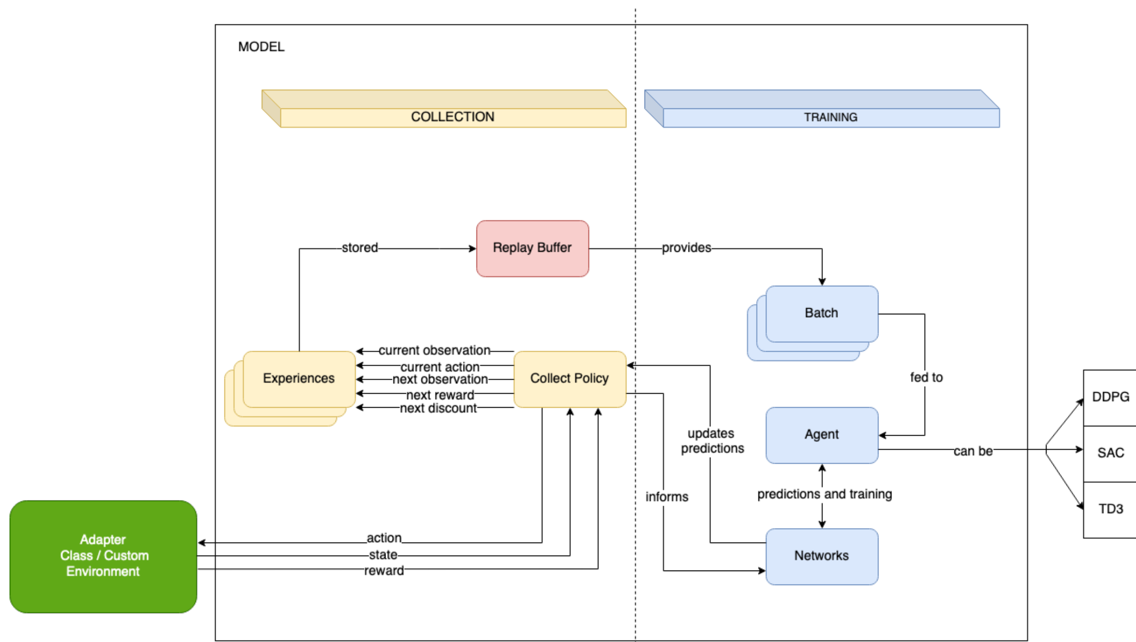


Figure 18: Model architecture.

The knowledge gathered by the agent is stored in the weights of the neural networks. The number of input nodes of the neural networks must match the environments number of state variables, and the number of output nodes must match the number of possible actions.

This is done dynamically since in OPAL it can be altered the number of the elements in the vector that contains either the state or the actions variables in the Driver and it will automatically change in all the models created without hard coding it in the models.

3.3.3 *Decisions*

OPAL is tested by implementing three different agents, but a lot more models can be implemented with this architecture.

To test them fairly, in terms of choosing the hyperparameters of each neural network and respective configurations, the three of them are tested with the same conditions.

For OPAL, and in special for pH1, the reinforcement learning problem is defined as follow:

- **Action:** An array where each element of the array is a metric that can be changed in the middleware (e.g. garbage collector)
- **Observation:** An array where each element represents the current state of a metric in the middleware and in the monitoring system (e.g. latency, etc.)
- **Reward:** The objective is to get as better performance as possible (e.g. higher throughput as possible).

In terms of the action chosen, even though there could be a lot of tunable parameters (the action given to the Driver from the agent is composed by a vector and not only one parameter), the time it takes to call the garbage

collector is the most crucial one for this specific case, as explained before. Without OPAL, the garbage collector is programmed to be called on their own, when the system thinks it is appropriate. Our action has the interval from 20 to 40, since different workloads, can in fact have a different optimal time to call the garbage collector.

For pH1, in terms of the state, the parameters that would show a significant change in the middleware were chosen, as shown in Table 2.

Metrics	Description
YCSB Throughput	Number of operations per second the system was doing.
TSO logs Throughput	Number of operations per second that corroborate the ones from YCSB.
Read Latency	Average read latency of the system obtained after taken an action.
Update Latency	Average update latency of the system obtained after taken an action.
Total Keys	The number of keys existing in the cache.
Total Requests	The number of keys that were consumed in the cache and in the database.
Total Cache Hits	The number of keys requested that were actually in the cache.

Table 2: Set of elements that compose the state in the reinforcement learning process.

In pH1, our reward consists on the throughput obtained from the middleware since it is continuously harvested during the optimization process and the reward consists in a numerical score based on the state of the environment. For the case of a continuous state space, for the agent to learn easily, the reward should be continuous and differentiable.

Despite pH1 being our use case, there were some preliminary tests done to try to guide the construction of OPAL in the best way possible, where the action, the observation and the reward change, because the use case also changes. For these preliminary tests, the action, observation and reward will be explained when they are presented in Chapter 4.

Hyperparameter	Value
Number of Iterations	1000
Actor Learning Rate	0.0001
Critic Learning Rate	0.0001
Replay Buffer Size	100 000
Batch Size	256
Discount Factor	0.99
Initial Collection Steps	1
Collection Steps per Iteration	1
Train Steps per Iteration	300
Number of Episodes for Evaluation	50
Evaluation Interval	100

Table 3: Actor-critic hyperparameters (baseline).

The hyperparameters defined for the actor-critic algorithms are shown in Table 3. These hyperparameters are defined as a start off, and as the experiments were done, they were changed accordingly to the results obtained.

Analyzing the table, the number of iterations is the number of episodes of training the agent is going to do. The learning rate controls how much to change the model in response to the estimated error each time the model weights are updated. Both the actor and the critic learning rate are set to the default value. The optimization

function (the function used to update the neural networks weights) chosen was the Adam optimizer to apply the gradients to the model parameters. Adam is the most common optimization function and was chosen because it can handle sparse gradients on noisy problems (and pH1 is a system prone to tremendous noise). The replay buffer size is the maximum length of the replay buffer. It is kept on the basis of memory available. The batch size is the size of a batch of experiences that are fed to the agent. The discount factor parameter is set to the default value, 0.99. This means that a future reward is valued as much as an immediate reward. The collection steps per iteration is the number of collect steps the agent is going to do per iteration of training. It is set to 1 so there is only 1 step of the collection at the onset of training. To keep the model from continuing on for infinitely many steps there is a limit the number of train steps per iteration to 300. The number of episodes to evaluate was set to 50 and these episodes were evaluated after 100 iterations of training (evaluation interval).

Step Method

In the step method there is a need to differentiate when an episode ends from when an episode continues performing steps. Since the agent is being tuned online, there is no concrete goal (it is only important to maximize the performance as much as possible). With this in mind, it is not possible to end an episode when the goal is achieved (like it is done in video games) because there is no knowledge of what is the goal to be achieved (the best performance the system can have) and that goal can change if the workload changes for example.

Instead the episode only ends if there is a condition that is not being met, for example, if the train steps per iteration are already at 300 (chosen value from Table 3). Adding to that condition, it was added another one in order to make the agent learn faster, that can be seen below:

Algorithm 1 Episode condition

if $all_time_reward >= current_reward$ **then**
  return termination$(state, reward = 0)$
else
  $all_time_reward \leftarrow current_reward$
  return transition$(state, reward, discount = 0.99)$
end if

This condition is obtained by first creating a variable, all_time_reward, that during the learning process will have the best reward achieved from all time rewards. Then, it is compared the current reward the model obtained with the all_time_reward variable. If the current reward is lower or equal to the all time reward, then the episode is terminated and the reward is 0 so the agent knows that there is no good in choosing that setup, it already has a better one with the all time reward.

If the condition is not met and the episode did not end, the transition method is called where the current reward is passed as a parameter so that the agent can understand how good it was the action chosen. The discount factor (already defined) is passed as the other parameter and it is taken into account the state of the system so that the agent knows that future reward is as important as immediate reward.

All of this will help the system understand if it is getting closer or not to an optimized state.

Online Tuning

We propose OPAL as a mechanism that can optimize a system in a realistic online environment with unstable performance and noisy data. In this online environment workloads can change at any time and crashes can happen in the middleware system, but OPAL will still learn from it and try to optimize the system.

The Driver component in OPAL can help with this online problem because it is built in a way that allows for the systems to provide the necessary information without having to restart our solution in any circumstance and more than that, our solution allows for the user to implement the environment to be optimized in any way the user desires. This means that it is the user's responsibility to implement the Driver methods in a way where in the middleware/system side the connection is never broken and the middleware never crashes.

Although there are several research projects for automatic system tuning [1, 2, 3, 4], they have not yet fully addressed he challenge of online tuning with the exception of the Multi-Model Database Tuning System [4]. In this paper, their approach can lead to possible drawbacks already mentioned in Chapter 2.3.4, making us opting to focus in the models themselves and configure and tune their hyperparameters in order for them to learn faster and adapt to other situations instead of using multiple models that compete against each other.

With online tuning in mind it is used the epsilon greedy algorithm in our policy. Actions are chosen at random according to a value epsilon which varies between 0 and 1. If epsilon is set to, for example, 0.1 then actions will be chosen at random 10% of the time, otherwise the best action outputted by the neural network is chosen. This allows for the algorithm to try unexplored options in case they are actually better than the action the neural network chooses from previous knowledge, avoiding getting stuck on local optimums (exploitation vs exploration). With this the algorithm will learn faster and be able to adapt better to different circumstances (workloads).

It is also used the experience replay technique that is incorporated in the DDPG, SAC and TD3 for online training. In fact this is the only method that can help with the online process by generating uncorrelated data. This can be achieved by reusing experiences where the learning agent simply remembers its past experiences and repeatedly presents the experiences to its learning algorithm as if the agent experienced again and again what it had experienced before. The result of doing this is that the process of credit/blame propagation is sped up and therefore the networks usually converge more quickly.

Although experience replay can make the networks converge faster, there is a need to pay attention that with this method past experience may become irrelevant or even harmful if the environment changes. Since the focus was to use different workloads online, this method at first was disregarded. But since it is such a powerful method for the networks to learn faster, it was decided to find the right balance in terms of how many experiences would be in the replay buffer. The experiences in the replay buffer should always be the most recent ones, exactly for the fact of a change in the environment. With this OPAL can get the benefits from the experience replay when the buffer still has experiences from the environment not changed and when the environment changes, it will take a short amount of time for the experiences to also change in the buffer, since the buffer should have the most recent ones and discard the old ones.

Considering that OPAL is being used to tune a system that changes its workload after 1000 iterations, the following goals should be achieved:

1. The system should not get stuck in a local minimum/maximum (epsilon greedy algorithm helps with this).

2. The networks should converge as fast as possible to the optimal solution (experience replay can help with this by selecting the experiences in a random way).

3. The system should not get lost with a large number of experiences, especially harmful experiences from a previous workload (it can be allocated a number of experiences to the buffer, for example 100, and guarantee these ones are the most recent ones, so that when the environment changes, these experiences change too).

By getting a batch of experiences in a random way, there is a possibility, when there is a workload change, for the agent to randomly pick the oldest experience that was produced with the old workload. With time, the probability of this happening reduces significantly as long as the number of experiences kept in the replay buffer is not too high, but enough to make the learning faster.

It is worth noting that Experience Replay is sensitive to the Replay Buffer size, with huge performance drops for buffers too small or too large (specially when using different workloads). This was taking in consideration when analyzing the results of the first preliminary tests and the buffer size was adapted to the value that gave us the best results.

4

SYSTEM ANALYSIS AND PERFORMANCE EVALUATION

To evaluate OPAL, two types of tests were made: the preliminary tests and the pH1 tests. Both types are important in their own way to have a better understanding of how OPAL works and can achieve good performances on the problem at hand. The first tests were used to validate the model in a more controlled situation, and the second tests were used to test the model in a real world scenario.

4.1 PRELIMINARY TESTS

The preliminary tests were conducted to determine the effectiveness of the various models selected for this case, to compare their performances, and to ensure that OPAL architecture is capable of handling other optimization problems of this nature.

In light of this, the preliminary tests involve simulations of several scenarios that can serve as examples of a pH1 environment by applying diverse use cases (mathematical functions). For example, it can be simulated a case of trying to figure out a simple function maximum, to understand how OPAL behaves in this specific scenario (e.g., maximize the performance of a system).

4.1.1 *Experimental Setup*

The preliminary tests evaluate OPAL in a dedicated machine (Dedicated Machine A) with 4 core *CPUs* and 2 threads per core. It is equipped with a 16 GB *RAM* and a SSD with 232 GB, where the operating system is *Ubuntu.* Its model name is *Intel(R) Core(TM) i3-4170 CPU @ 3.70 GHz* and its architecture is *x86_64*.

In order to quickly test alternative hyperparameters of the selected models or determine whether the model is truly going to converge or not and is acting as expected, several scenarios were generated where the environment is artificial. This is crucial because, from here to a real online system (pH1), which takes a very long time to change configuration and has a lot of noise, these tests can provide a certain assurance into claiming that the models used and the OPAL architecture are strong and reliable sources for the foundation needed to support the optimization of a middleware system.

Despite the fact that the plan is to compare 3 reinforcement learning algorithms, in the preliminary tests there is only one of them that is gonna be evaluate in all the scenarios created.

DDPG is the algorithm to be tested in these scenarios because it is the simplest one out of the ones to implement and understand and because the others (SAC and TD3) have the same structure basis as DDPG,

only differing in certain additional features already explained in Chapter 2.2.3. Only choosing DDPG, was a decision made in order to have a fair comparison between all the scenarios. In other words, this decision was made to not have any scenario accused of getting better or worse results in a faster or slower way because there were different algorithms compared. The results from the scenarios tested should only differ because of the environment chosen for each one so it can be understood how hard it is for the model to achieve a good performance as the scenarios get more and more complicated.

The first tests were on simpler functions and granularly it was increased these functions complexity. For these scenarios the action was the variable x and the threshold given to this action was the interval from 0 to 1. The observation and also the reward in this case was the function result, y, with the threshold also between 0 and 1.

As shown in Figure 19, it was created a Driver Simulator in Machine A for preliminary tests that implements the appropriate actions (changes the variable x for the functions) and also processes the state (result of the function) and calculates the suitable reward (maximize the functions).

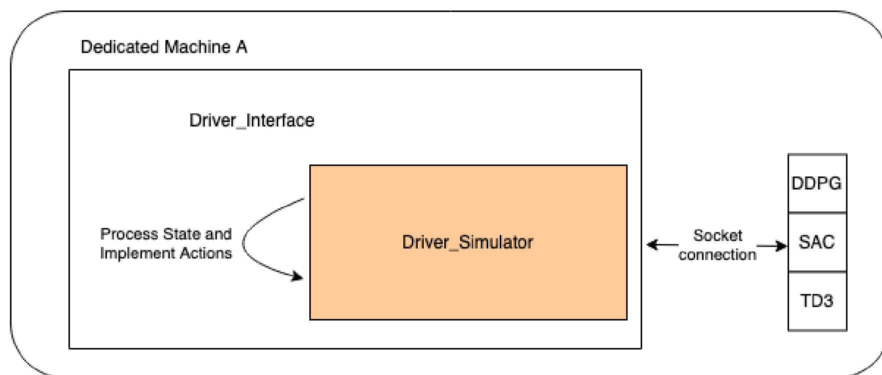


Figure 19: Wrapper for simulations.

Scenario A

The first scenario is chosen for its simplicity. If OPAL is optimizing a system where its complexity is linear, for example a system where when increased the time for the garbage collector to collect, the performance of the system will also increase in the same way, the result is one of the simplest systems for OPAL to tune: the function $y = x$ shown in Figure 20.

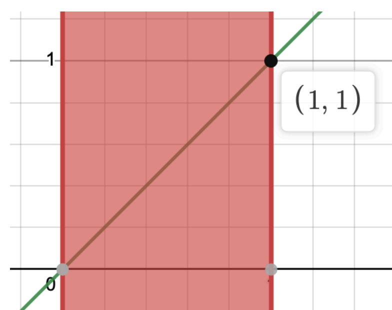


Figure 20: Function $y = x$.

This is an important scenario to be tested because, if at first, the model is not well configured, with this scenario, it is easy to instantly recognize the error and fix it before moving forward. Also, it is one of the best

scenarios to help with hyperparameter tuning. Since it is so simple, when trying to optimize a parameter it can be observed how that parameter will affect the model and its performance without having any complexity layers that can make the learning process harder to identify.

Scenario B

Usually real world systems are not quite as simple as the previous scenario. Transactional systems have noise, delayed results, crashing situations, and so on.

In pH1, for example, even if the system is not changing its configurations, at every iteration, latency is changing up and down. This is due to how the underlying database system considered in pH1 (Cassandra) is built. There is going to be a different latency when only a read workload is used, or only an update workload is used or even if it is used a mix workload (50% reads and 50% updates). Furthermore, in a read workload, the latency may even have small variations due to the noise caused by the database and the middleware.

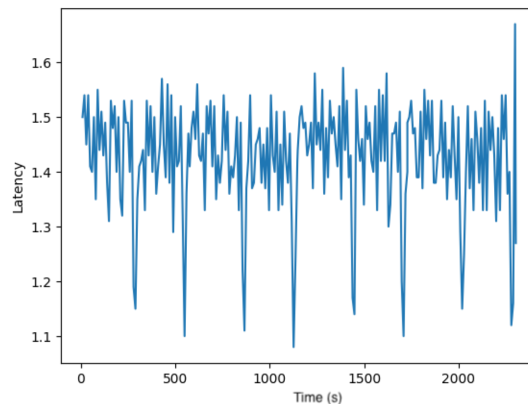


Figure 21: pH1 latency variation when populating Cassandra.

In Figure 21, pH1's latency variation is represented, without any optimization, during Cassandra populating process. A constant function from the figure as $y = 1.45$ can be extrapolated and it can be taken into consideration that the noise is responsible for the points that do not match that function. This means that normally the latency for pH1 when population Cassandra is 1.45 with some added noise and outliers for crash/delayed moments.

OPAL was built to be able to handle this noise and other kinds of unexpected events, like high peaks and harsh crashes with certain mechanisms already explained (e.g., replay buffer to maintain certain memory). To test this, the second scenario uses the same function as Scenario A, but it is added a certain noise (with a noise function) to it, as shown in Figure 22.

The noise function receives a coefficient of noise that will impact how much of the function value should increase or decrease at a certain point. If the coefficient is for example 0.2, then the noise added is a decimal number randomly selected between the interval from -0.2 to 0.2.

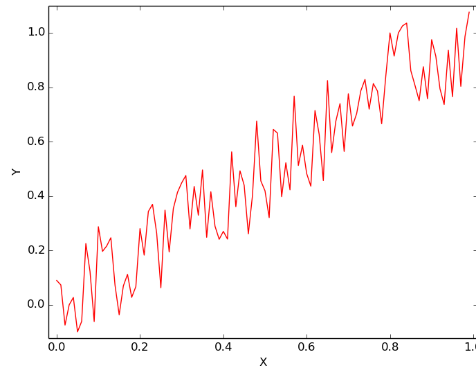


Figure 22: Linear function with a noise coefficient of 0.2.

The decision of using 0.2 falls into the conclusion taken from doing some tests with pH1, for example when populating Cassandra in Figure 21, where a latency variation around 0.2 can be seen in the constant function where $y = 1.45$.

Scenario C

OPAL is built to handle complex systems.That said, the next incremental step was to create a scenario that can get close to a complex system. A complex function is subjective and there is a range of different functions that can be considered complex. The function $y = \frac{sin(8 \cdot x)}{2} + x$, described in Figure 23, was chosen to achieve that complexity.

It was used the sine function in the domain [0,1] because of its periodicity, which gives us a chance to relate with pH1 when the same workload is run consecutive times, making it a periodic system.

It was added to the sine function the addition, multiplication, and division operations to provide a function with a local minimum and a local maximum and an absolute minimum and an absolute maximum.

Having different maximums (or minimums) is a common situation in middleware systems because sometimes their own performance can be high with a certain configuration but extremely high with another.

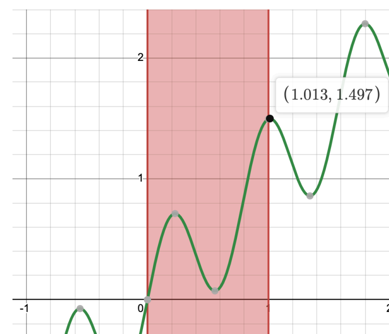


Figure 23: Function $y = \frac{sin(8 \cdot x)}{2} + x$.

Scenario D

In the last scenario, it was taken a step further from the previous scenario and added noise to the complex function $y = \frac{sin(8 \cdot x)}{2} + x$. This noise comes to help in the same way as in scenario B, which means it was added with the intention to create a more realistic function that could better simulate a real scenario, taking into account that these systems, in particular pH1, have a lot of noise.

It is shown how this noise will affect this function in Figure 24.

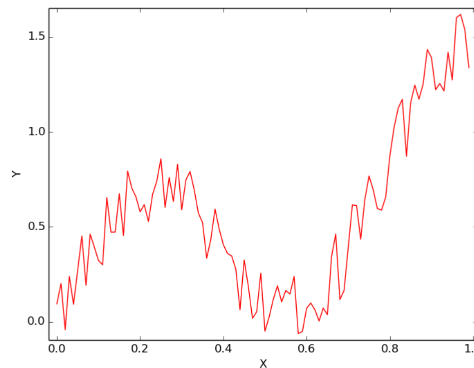


Figure 24: Complex function with a noise coefficient of 0.2.

4.1.2 *Results*

To compare the performance, it was measured the average return and the loss each scenario obtained using the DDPG model.

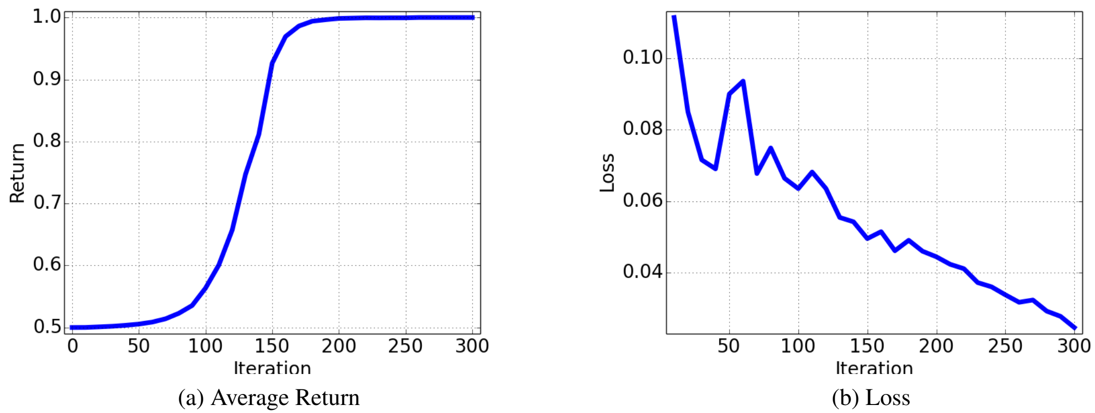
The average return is the simple mathematical average of a series of returns generated over a specific period of time. This return is the reward gained from the model during the evaluation periods, over 10 episodes.

The loss is the penalty for a bad prediction. If the model's prediction is perfect, the loss is zero; otherwise the loss is greater. The loss seen in the next results is a combination between the critic and the actor loss in the DDPG model, because both of them are both important to evaluate the algorithm. Critic loss has an inverse correlation with actor loss, so it is important to consider both of them and understand that in this model when the critic loss decreases, the actor loss increases, making it a necessity to find a good balance between both.

Scenario A

In scenario A, the result expected in terms of average return would be 1.0, since the goal is maximizing the function $y = x$, within the domain [0,1], and assigning the reward as the state of the model.

It is shown in Figure 25 (a) that indeed the average return of the model converges to 1 in less than 300 iterations. The loss graphic in Figure 25 (b) can also corroborate the good results since it shows a significant decline since the beginning.



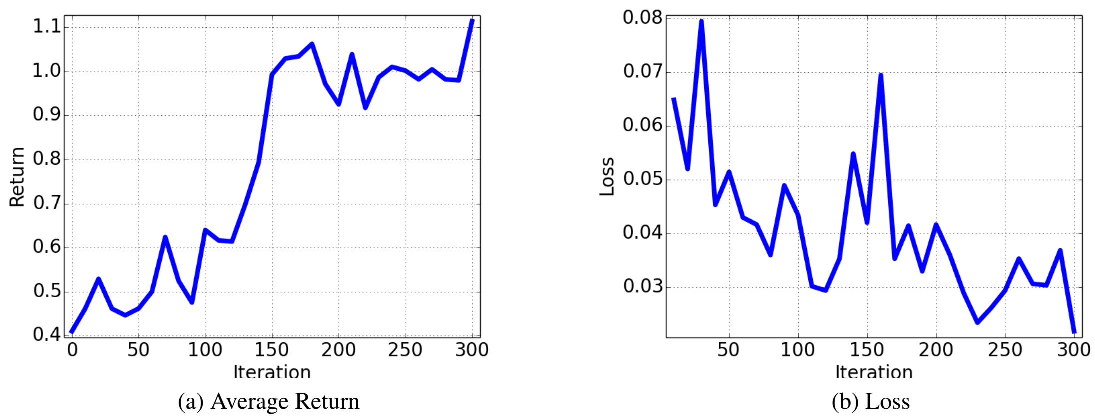
(a) Average Return (b) Loss

Figure 25: Performance of scenario A.

Scenario B

In scenario B, it is shown in Figure 26 (a) that the average return also converges to 1, but with some more difficulty. This is normal and expected from the results of a scenario where noise is introduced.

The loss also decreases considerably, as shown in Figure 26 (b), but with some low rises and falls that should also be blamed on the introduction of noise.

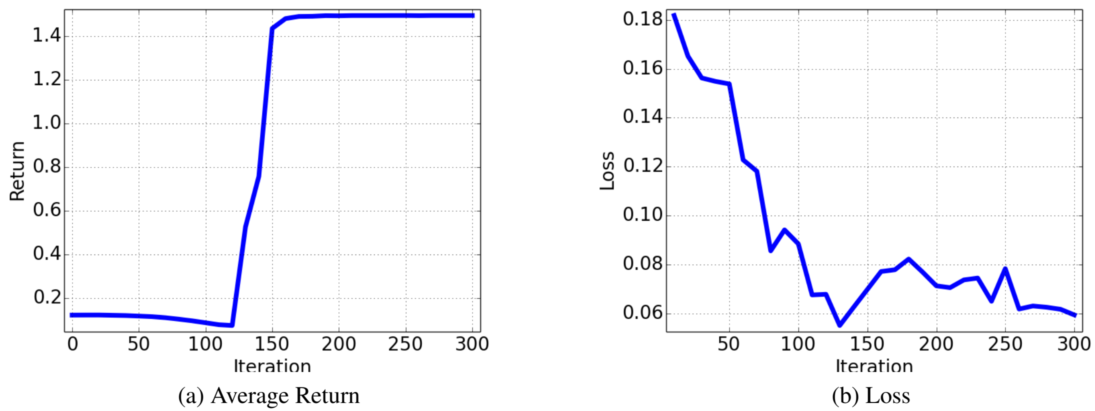


(a) Average Return (b) Loss

Figure 26: Performance of scenario B.

Scenario C

According to Figure 23, the model in this scenario should converge to 1.49 in terms of average, since it is the function absolute maximum. The results in Figure 27 (a) are very good since indeed the model converges to 1.49.



(a) Average Return (b) Loss

Figure 27: Performance of scenario C.

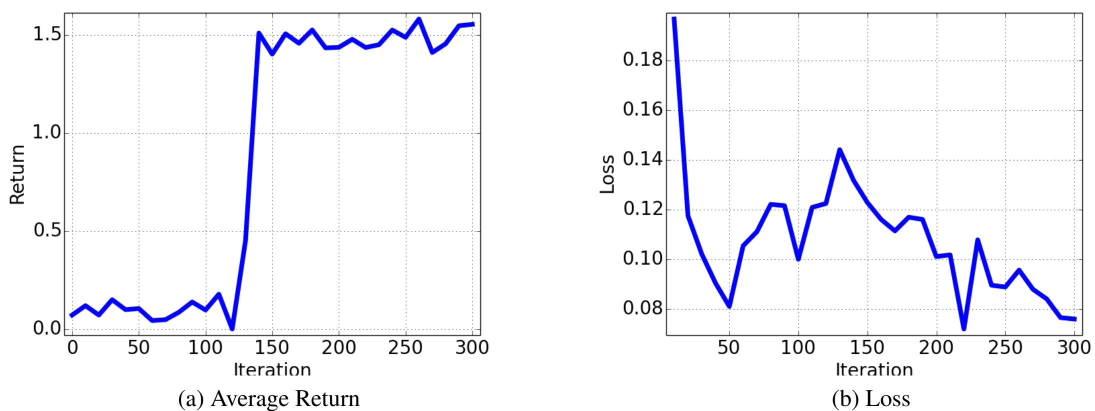
The loss in this model, that is represented in Figure 27 (b) decreases with time, but there are some bigger fluctuations compared with the loss of a simpler model (e.g. scenario A). This is due to the function's periodicity and the fact that this function has two different maximums.

Scenario D

The noise function coefficient for scenario D can also add a certain quantity of noise to the absolute maximum. So, there is a possibility of falling into a scenario where the return would be more than 1.49. That can be seen in Figure 28 (a) in iteration 150, where the average return is bigger than 1.49.

With the addition of noise in the scenario D, the model still can converge into the expected return value, with the loss decreasing as shown in Figure 28 (b). This proves that the model can find a solution to even a complex function with noise.

It is also worth noting that after analyzing this scenario (and the other previous three), the optimal result is always obtained in the first 300 iterations. This would be helpful to considerate how many iterations OPAL will need to improve the performance of pH1.



(a) Average Return (b) Loss

Figure 28: Performance of scenario D.

4.1.3 *Discussion*

During the previous scenarios, with DDPG, the hyperparameters of the model were tuned according to the performance and loss, achieving the best results with the following configurations in Table 4. These values were chosen from this point on as the base configuration for the comparison between models.

Hyperparameter	Standard Value	Optimized Value
Number of Iterations	1000	300
Actor Learning Rate	0.0001	N/A
Critic Learning Rate	0.0001	N/A
Replay Buffer Size	100 000	1000
Batch Size	256	64
Discount Factor	0.99	N/A
Initial Collection Steps	1	N/A
Collection Steps per Iteration	1	10
Train Steps per Iteration	300	100
Number of Episodes for Evaluation	50	5
Evaluation Interval	100	50

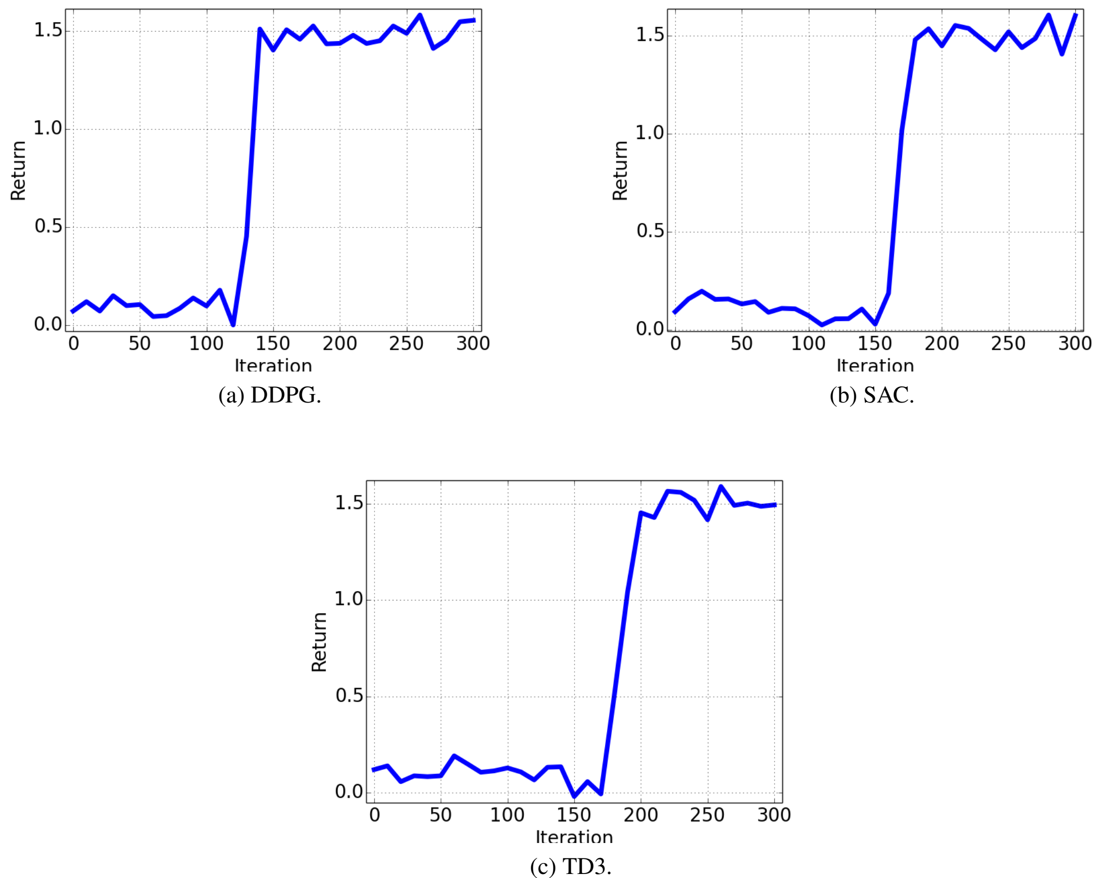
Table 4: Hyperparameters tuned (improvement).

As shown in Table 4 some of the hyperparameters defined before in Chapter 3 were modified because better results were achieved when tuning them in the preliminary tests.

The number of iterations in a production machine should be infinite since OPAL would always keep running in an online environment waiting for a change in the problem so it could adapt its optimization model. To be able to perform tests and show results in a relatively short amount of time it was decided to reduce the number of iterations to 300 because from the preliminary tests the results showed that a lower number of iterations of training (300) would not affect the performance of DDPG. The replay buffer size was also changed to a smaller number. The perfect number to achieve a balance between how many experiences should be in the buffer for it to learn faster but still to not loose its effect when different workloads (or crashes) are introduced was 1000. When the size of a batch of experiences which are fed to the agent was reduced to 64, there was also a very big improvement in terms of time. The agent would learn and converge with a lot less iterations. This happens because small batches go through the system more quickly and with less variability, which fosters faster learning. The reason for the faster speed is obvious. The reduced variability results from the smaller number of items in the batch. It was decided to collect 10 steps per iteration, instead of 1. This change improved the average return metric. This is due to the collection steps being able to provide the agent more information about each iteration using the replay buffer. The number of train steps per iteration was limited to 100, instead of 200 since 100 steps was enough for the agent to learn with the information provided in that iteration. The number of episodes to evaluate was set to 5 because it was enough to understand if the agent was learning correctly or not, evaluating these episodes after 50 iterations of training. The evaluation interval was also reduced because the number of iterations had been reduced too.

Since scenario D is the most complicated one studied, it was the one chosen to compare DDPG with SAC and TD3 using the hyperparameters from Table 4.

Figure 29 shows the three different methods and compares them in terms of the average return they can provide.



(a) DDPG.

(b) SAC.

(c) TD3.

Figure 29: Comparison between different models.

The results obtained show that all 3 methods can achieve good results. Despite this, it is DDPG that can achieve those results the fastest (with the least number of iterations). SAC and TD3 start only converging after the 150 iteration and DDPG starts converging in the 120 iteration. This can happen because TD3 and SAC try to handle different problems of DDPG (and those solutions are not needed for the problem at hands) and that can increase the time to find the optimal solution.

For optimizing pH1 and other similar systems, the time a model takes to converge in a good result is crucial, since an online system should not take a lot of time (iterations) to tune its parameters and deliver a performance with high quality. For that reason, it is as important to reduce the time a model takes to find a good solution (achieve the lowest time possible) as it is to really find the optimal solution.

The results obtained provided the confidence necessary in OPAL to try use it for the optimization of pH1, and specially use DDPG to increase performance in the shortest time.

4.2 PH1 TESTS

pH1 tests were conducted in order to use the middleware as a use case to understand if OPAL could indeed optimize a transactional system. Even though pH1 is seen as one of many use cases for this project, the achieved results provide confidence it can be replied to any other system of this kind.

4.2.1 *Experimental Setup*

To use pH1 middleware, the version of Cassandra has to be equal or lower than 2.0 because this middleware was implemented with these versions.

A load injection mechanism, YCSB [48] is used for the evaluation of the system. YCSB allows for the creation of a benchmark that changes over time, so that the dynamic (continuous adjustment) component of the system can be evaluated.

The Yahoo Cloud Service Benchmark is a benchmark that allows the comparison among data stores designed for the cloud computing paradigm. In other words, Yahoo! developed this benchmark because both the paradigm and the access pattern of such data stores are quite different from the ones used by traditional benchmark systems, mostly designed for relational databases.

The YCSB benchmark starts by creating a defined number of concurrent clients that will try to perform a set of operations according to a pre-defined workload. The type of operations available include Read, Scan, Delete and Update operations. Each workload results from a single operation type, or a composition of all.

For pH1, two dedicated machines were used to test OPAL.

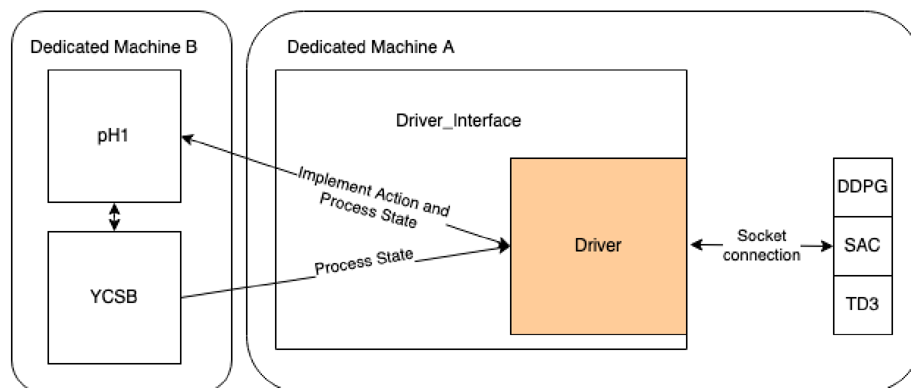


Figure 30: Wrapper for pH1.

Machine A has the same configuration and setup as the Machine A described in the Preliminary Tests section. It is in this machine that OPAL is deployed with a Driver that is adapted to connect with pH1 and YCSB.

Machine B has the pH1 middleware that is connected with the underlying database (Cassandra) and YCSB. This machine has the same configuration as Machine A from the Preliminary Tests and connects with the Driver from Machine A through ssh connections.

As explained in Figure 30, the Driver is configured to be able to connect and implement actions in pH1 and to obtain the necessary metrics from the YCSB and pH1.

To evaluate OPAL, each test requires the following steps:

1. Choose and run the model in the Learning module of OPAL, so it can open the socket that will be listening to connect with the other parts of the system.
2. Run the Timestamp Oracle that will connect with pH1 so it can have the timestamps that will guarantee the transactions are correct.
3. Run the YCSB benchmark that calls pH1.
4. Run the Driver in the Wrapper module of OPAL so it can connect with the python part of OPAL and with the middleware and the monitoring system.

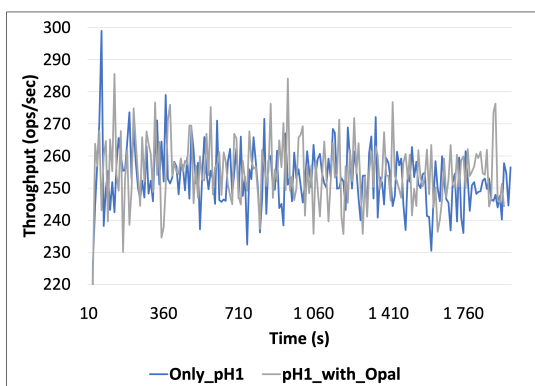
4.2.2 *Results*

Four different benchmarks were used to test OPAL. The first one consists on a read-only workload, using 1 thread and 1 000 000 operations. The second one is a mix workload (50% reads and 50% updates), with also 1 thread and 1 000 000 operations. The third benchmark is built with the same configuration as the first, one but with 5 threads and 5 000 000 operations (since the number of threads was increased). Finally, running two different workloads, one after the other was also tested. This allowed to understand if OPAL can still improve when a change in the scenario takes effect.

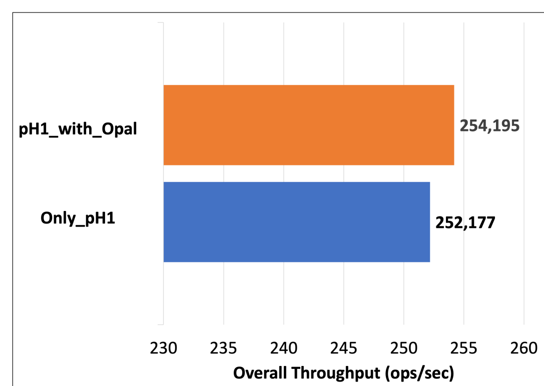
To evaluate these benchmarks, the average throughput is considered throughout the time a workload is running, so it can provide a better idea of the performance of the system. Moreover, the overall throughput in pH1 with and without OPAL was compared, as to visualize the improvement in pH1 when using OPAL. On mix workloads, different latency patterns may occur, concerning only reads or only updates. These plots are important in terms of results since it can provide a different view about how the system is truly behaving.

Read Workload

The first test was with a read-only workload to simply test the system in terms of reliability and implementation.



(a) Average throughput in pH1 with and without OPAL.



(b) Overall throughput in pH1 with and without OPAL.

Figure 31: Throughput in a read workload.

It is expected for OPAL not to have a significant improvement when testing with a read only workload, since the action (the time to call the garbage collector) does not directly interfere with the proper functioning of pH1 in a read environment.

As it can be seen in Figure 31, (a) it is indeed very difficult to distinguish between the performance of pH1 with or without OPAL and, that the overall throughput is affected, as it shows the same behavior in both cases (Figure 31 (b)). Figure 32 corroborates the other two graphics shown before, as it can also be seen that the average latency does not improve.

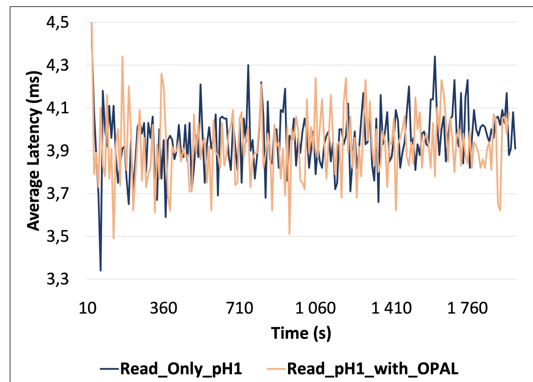


Figure 32: Average read latency in a read workload in pH1 with and without OPAL.

These results meet what it is expected since the cache in pH1 serves to store intermediate versions that are written by other transactions. If the workload is read-only, then the number of cached reads is reduced, as they go directly to the database.

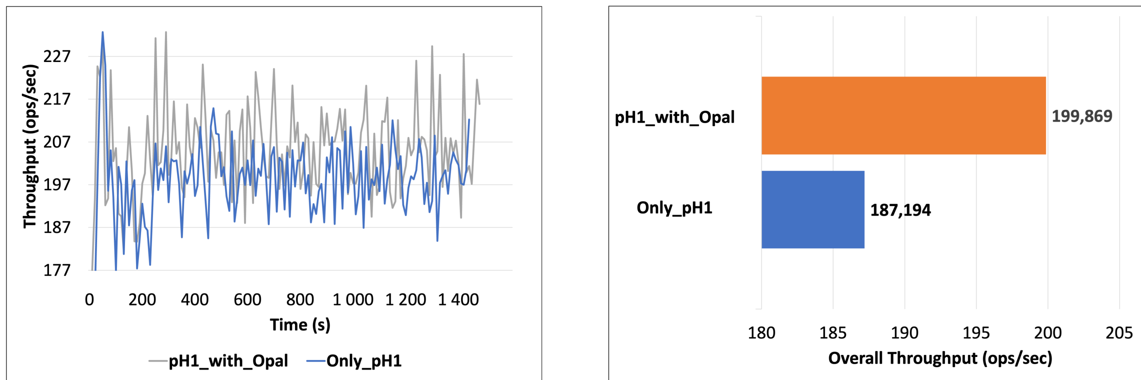
The architecture of pH1 is very efficient in terms of reads since it does not have to save intermediate versions in the database. This is the reason why there is almost no difference when using the garbage collector and because of that there is also no difference in using or not using OPAL.

Mix Workload with 1 thread

This test considers a mix workload, assessing if there is any effect OPAL could offer to a complicated workload that does not behave similarly throughout the entire time it is running; as occasionally, it has bursts of reads or update operations. The performance obtained is shown in Figure 33.

As it can be seen in Figure 33 (a), the average throughput is constantly higher when using OPAL than without it. It is still very connected with the performance of only pH1 (no OPAL). As it can be seen at the beginning of the workload, both lines overlap frequently, but then with time, the gray line is constantly above the blue one. This proves OPAL is improving and trying to achieve better results.

In this workload, pH1 achieves a 6,34% better performance with OPAL, as it can be seen in Figure 33 (b). This is considered already very good results since this system has a lot of noise and it is only using one parameter to tune.



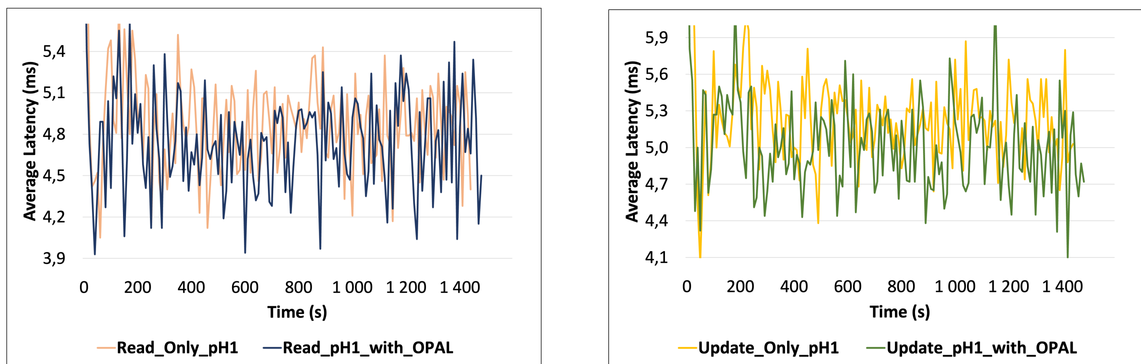
(a) Average throughput in pH1 with and without OPAL.

(b) Overall throughput in pH1 with and without OPAL.

Figure 33: Throughput in a mix workload (50% reads and 50% updates).

The average read latency graphic (Figure 34 (a)) normally shows a lower latency with OPAL (dark blue line) than without it (orange line). This also happens in the update latency (Figure 34 (b)) with the dark green line.

Despite these frequent results throughout the workload, there are some outliers in terms of the OPAL performance in Figure 34 (a) at 430 and 1000 seconds where the read latency is lower without OPAL. It can also be shown in Figure 34 (b) at 210, 615 and 1100 seconds where the latency with OPAL is higher than without it or at 500 seconds where the latency without OPAL is lower.



(a) Average read latency in pH1 with and without OPAL.

(b) Average update latency in pH1 with and without OPAL.

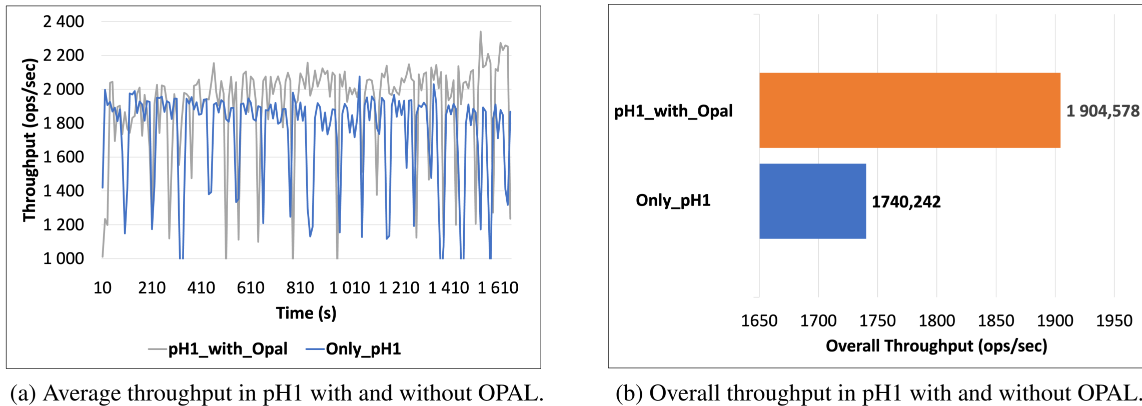
Figure 34: Average latency in a mix workload (50%reads and 50%updates).

The outliers are disguised in Figure 33 (a) since the throughput graphic does not distinguishes between reads and updates. The noise of Cassandra and crashes of pH1 are responsible for these outliers when using the Timestamp Oracle and when memory gets full.

Mix Workload with 5 threads

This test compares how OPAL can make a difference in terms of using Cassandra with pH1 during a mix workload with more than just one thread. Using more than 1 thread would make tuning the time of the garbage collector to take action more important and crucial, since there would be more load in the system.

The performance obtained is shown in Figure 35.

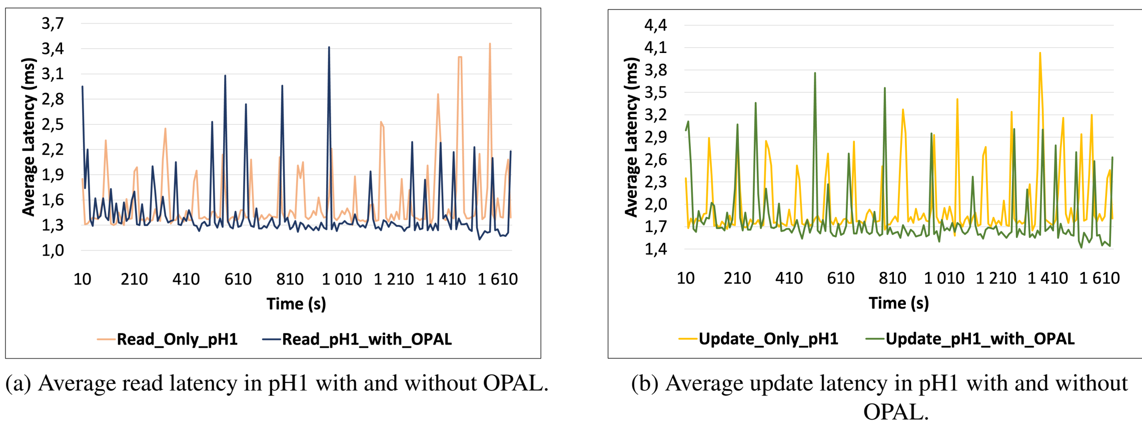


(a) Average throughput in pH1 with and without OPAL.

(b) Overall throughput in pH1 with and without OPAL.

Figure 35: Throughput in a mix workload with 5 threads.

An improvement is depicted in Figure 35 (a) when compared with Figure 33 (a), since the average throughput shows a very significant increase with time when pH1 is using OPAL (grey line). The overall throughput in pH1 with OPAL also shows an increase of 8,63% compared with not working with OPAL Figure 35 (b).



(a) Average read latency in pH1 with and without OPAL.

(b) Average update latency in pH1 with and without OPAL.

Figure 36: Average latency in a mix workload with 5 threads.

Figure 36 shows a constant decrease in latency that confirms the increase in terms of operations per second shown in Figure 35.

With 5 threads, there are a lot more of outliers, not in terms of the OPAL performance as shown in the previous use case, but in terms of peaks (latency) and lows (throughput). As an example, on Figure 35 (a) at 410, 610, 650, 810, 1010, 1210 and 1410. These outliers exist when using OPAL and also when not using it, concluding that they are due to synchronous problems with threads when a lot of memory is used and not because of OPAL, since the same workload but only with 1 thread have less outliers like this Figure 33 (a).

These results corroborate that OPAL can in fact help optimizing a middleware system. It can be seen the obvious benefit in this scenario, since the load is bigger, making it easier for OPAL to learn better in the same time frame.

Different Workloads

OPAL was assessed by running different workloads over a period of time and verify if it could learn to improve them.

In order to change workloads the state array (given to the model) had to be modified. Different workloads can provide different state parameters or not even providing a state parameter in YCSB. For example, a read workload does not have an average update latency parameter, but an update workload or a mix workload do. To be able to work with all parameters, a state array was installed with all the variables and when in one workload the parameter does not exist, in the array the variable will be the number 0 to let the model know it can not learn with that parameter. This will not make the model think that the previous action was a bad action and that is the reason that the parameter for that state is 0 since after that action. All the other actions in that specific workload would also give 0 as the state to that parameter no matter what. On the contrary, this is beneficial because it can be a good way for the model to learn that a different workload is being used and it should start exploring more than exploiting.

More specifically, an update-only workload was applied, followed by a mix workload. The configurations for both these workloads are the same as for the previous one (with 5 threads). The results obtained are shown in Table 5.

	WITH OPAL			WITHOUT OPAL			GAIN
	Throughput	Read Latency	Update Latency	Throughput	Read Latency	Update Latency	Throughput
Update Workload	1555,925	-	2,0249	1387,619	-	2,244	**+10,8%**
Mix Workload	1952,152	1,3638	1,7428	1643,926	1,597	2,062	**+15,79%**
Together	1820,077	-	1,837	1560,948	-	2,121	**+14,23%**

Table 5: Overview of the results obtained running a mix workload after an update workload.

The results show that the throughput without OPAL in a mix workload after running an update-only workload is lower than just running a mix workload by itself: 1743,926 in Table 5 against 1740,242 in Figure 35 (b). This is due to the fact that an update-only workload will in fact create a lot of garbage (write-set queues) in pH1, contributing to an increase in memory that can cause problems in the performance of the system if the garbage collector does not get activated in the right time.

To confirm that Opal can indeed help increase the performance of pH1 when tuning it in an online manner, the results also show that when running a mix workload after an update-only workload, it can be seen a gain of 15,79% in terms of throughput for the mix workload. This is more than the double gained from the previous mix workload with 5 threads that did not have an update workload running before it.

4.2.3 *Discussion*

Every workload ran in a time interval around 1600 seconds. The reason this time interval could not be extended was due to memory limitations and restrictions in the machines utilized. However, this interval was enough for the evaluation tests, since OPAL was created to adjust to different scenarios in less than 300 iterations.

In terms of a read workload, there was no significant results obtained. As explained before, this is due to the garbage collector not being utilized to obtain better performance in this type of workloads. These tests can be seen as a good example of a scenario where OPAL will no be able to improve the system, not because of the

system itself but because of the goals intended with the actions chosen. When implementing the actions in the driver, it is essential for the user of the system to understand the whole spectrum of how the action can or not affect the performance of the system trying to be optimized.

The results obtained with the mix workload with one thread show an improvement in pH1 with OPAL (6,34%). Despite this improvement, these results are not fully able to confirm the importance of tuning the time for the garbage collector to take place, since the system contains a lot of noise. The use of only one thread, also makes OPAL only slightly significant.

In order to prove the system works for pH1, OPAL was evaluated with 5 threads but with the same configuration as before. The results show an improvement of 8,63%, and confirm the solution can indeed help optimize a middleware system. The results are encouraging given that only one parameter (the most crucial one for pH1) was used.

Finally, it shows that the solution is indeed capable to be used for online tuning since it cab achieve sufficient results when running a mix workload after an update workload without having to restart the database or the middleware system for the action to be implemented.

5

CONCLUSION

This dissertation demonstrates how parameter tuning is a challenging problem, particularly for distributed systems. Its complexity results from the substantial number of variables and the intricate connections made between those variables and the performance of the underlying systems. New chances for self-tuning capabilities to be built into these systems are increasing as these systems get more complicated and the requirement for constant tuning grows.

The idea of dynamic parameter adjustment is not constrained to be used in database related systems, and any system which is subjected to variations in workload or even hardware capabilities should, in theory, benefit from continuous adjustment of its tuning parameters.

This dissertation studies the viability of using machine learning to optimize a middleware system in a black box, online perspective. The solution, Opal, is based on reinforcement learning, in particular, actor-critic methods in order to select the best set of strategy settings for these type of systems.

OPAL seeks to enhance the system's decision-making process by making it more dynamic and flexible. Additionally, it offers an adaptable and extendable framework to accommodate novel and various systems.

OPAL was implemented with two different modules with distinct concerns that interact with one another to leverage machine learning techniques, always with the intention to be extensible. It should be possible to add and change different rewards, observations and actions for the machine learning algorithm chosen.

It was tested using YCSB in 4 different benchmarks to evaluate the benefits and the overall cost of adding machine learning guarantees to the optimizer.

The results show an increase in performance when tested with a mix workload with 5 threads: 8,6% gain in terms of throughput. When testing different sequential workloads (mix after update) it can be confirmed that Opal is able to be used online with a performance up to 15,79% compared with not using Opal after all.

Since the middleware has unpredictable characteristics and a lot of noise, where the only parameter changed is the garbage collector (despite being the most critical one), the results are very satisfactory taking into account all the conditions faced, in particular, the difficulty in optimizing a system composed of many others: pH1 itself, the Timestamp Oracle and Cassandra.

5.1 FUTURE WORK

One of the main goals for this dissertation was to create a solution that would be applied to any middleware system. The prototype was implemented on top of pH1. However, it intended to test the solution on top of other

state-of-the-art transactional middleware systems. In short, a new research question would be to investigate whether the solution can obtain the same machine learning results and evaluate if its capabilities stays in the same order of magnitude as the one achieved for pH1.

Despite believing the interval of time used to evaluate OPAL was enough, running OPAL with pH1 in a higher time interval will allow to verify if the results obtained will continue to be in the same range as the ones obtained in these tests. Moreover, the increase in the load on the benchmark, by increasing the number of operations (more than 5 000 000) and the number of threads (more than 5) to induce higher load will allow to understand how the system behaves in such scenario.

More generally speaking, further tests should also be conducted with different reinforcement learning methods and different and/or more state and action variables. There is a belief that if the system had more than just one different continuous action, the system can perform better as long as the number of variables is not big enough for the agent to not be able to learn with them.

And finally, experiments evaluating the impact of dynamic variable adjustment should be conducted not just in the context of transactional middleware, but in the context of other systems, such as distributed databases, web servers or even big data processing platforms (e.g. Spark [50]), since OPAL is equipped for it.

BIBLIOGRAPHY

[1] Dana Van Aken, Andrew Pavlo, Geoffrey Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. pages 1009–1024, 05 2017.

[2] Ji Zhang, Li Liu, Minwei Ran, Zekang Li, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Wang Yangtao, and Tianheng Cheng. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. pages 415–432, 06 2019.

[3] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. QTune: a query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12:2118–2130, 08 2019.

[4] Yaniv Gur, Dongsheng Yang, Frederik Stalschus, and Berthold Reinwald. Adaptive Multi-Model Reinforcement Learning for Online Database Tuning. In *EDBT*, 2021.

[5] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.

[6] Sachin Salunkhe, D. Rajamani, Balasubramanian Esakki, and U Chandrasekhar. Prediction of life of piercing punches using artificial neural network and adaptive neuro fuzzy inference systems. *International Journal of Materials Engineering Innovation*, 10:20–33, 02 2019.

[7] Roohollah Amiri, Hani Mehrpouyan, Lex Fridman, Ranjan Mallik, Arumugam Nallanathan, and David Matolak. A Machine Learning Approach for Power Allocation in HetNets Considering QoS. pages 1–7, 05 2018.

[8] Amber. (Deep) Q-learning, Part1: basic introduction and implementation. https://medium.com/@qempsil0914/zero-to-one-deep-q-learning-part1-basic-introduction-and-implementation-bb7602b55a2c, 2019. [Online; Accessed: 2022-01-05].

[9] Jiang Fan, Zeng Yuan, Changyin Sun, and Junxuan Wang. Deep Q-Learning-Based Content Caching With Update Strategy for Fog Radio Access Networks. *IEEE Access*, PP:1–1, 07 2019.

[10] Fábio André Castanheira Luís Coelho, Francisco Miguel Barros da Cruz, Ricardo Manuel Pereira Vilaça, José Orlando Pereira, and Rui Carlos Mendes de Oliveira. pH1: A Transactional Middleware for NoSQL. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 115–124, 2014.

[11] David Reinsel, John Gantz, and John Rydning. Data Age 2025: The Digitalization of the World from Edge to Core. Technical report, IDC - International Data Corporation, 11 2018.

[12] Tom Mitchell. Does Machine Learning Really Work? *AI Magazine*, 18:11–20, 09 1997.

[13] Khushbu Kumari and Suniti Yadav. Linear regression analysis study. *Journal of the Practice of Cardiovascular Sciences*, 4:33, 01 2018.

[14] Mengqiu Kong, Dongsheng Li, and Daofang Zhang. Research on the Application of Improved Least Square Method in Linear Fitting. *IOP Conference Series: Earth and Environmental Science*, 252:052158, 07 2019.

[15] Joanne Peng, Kuk Lee, and Gary Ingersoll. An Introduction to Logistic Regression Analysis and Reporting. *Journal of Educational Research - J EDUC RES*, 96:3–14, 09 2002.

[16] Padraig Cunningham and Sarah Delany. k-Nearest neighbour classifiers. *Mult Classif Syst*, 54, 04 2007.

[17] Lior Rokach and Oded Maimon. *Decision Trees*, volume 6, pages 165–192. 01 2005.

[18] Jehad Ali, Rehanullah Khan, Nasir Ahmad, and Imran Maqsood. Random Forests and Decision Trees. *International Journal of Computer Science Issues(IJCSI)*, 9, 09 2012.

[19] Mahamed Omran, Andries Engelbrecht, and Ayed Salman. An overview of clustering methods. *Intell. Data Anal.*, 11:583–605, 11 2007.

[20] Huang Yi, Sun Shiyu, Duan Xiusheng, and Chen Zhigang. A study on Deep Neural Networks framework. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 1519–1522, 2016.

[21] Martin Puterman. Markov Decision Processes : Discrete Stochastic Dynamic Programming / M.L. Puterman. *Technometrics*, 37, 08 1995.

[22] baeldung. Epsilon-Greedy Q-learning. https://www.baeldung.com/cs/epsilon-greedy-q-learning, 2021. [Online; Accessed: 2022-01-06].

[23] Richard Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Adv. Neural Inf. Process. Syst*, 12, 02 2000.

[24] Haskell B. Curry. The method of steepest descent for non-linear minimization problems. *Quart. Appl. Math.*, 2:258–261, 1944.

[25] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The Elements Of Statistical Learning. *Aug, Springer*, 1, 01 2001.

[26] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[27] Hans Josef Pesch and Roland Bulirsch. The Maximum Principle, Bellman's Equation and Caratheodory's Work. *Journal of Optimization Theory and Applications*, 80:203–229, 02 1994.

[28] Michael McCloskey and Neal J. Cohen. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. *Psychology of Learning and Motivation*, 24:109–165, 1989.

[29] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting Fundamentals of Experience Replay, 2020.

[30] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-learning, 2015.

[31] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, 2018.

[32] scikit-learn developers. Factor Analysis process. http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.FactorAnalysis.html, 2007. [Online; Accessed: 2021-11-01].

[33] scikit-learn developers. K-means. http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html, 2007. [Online; Accessed: 2021-11-01].

[34] Robert Tibshirani. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58:267–288, 01 1996.

[35] C. Rasmussen and C. Williams. Gaussian Process for Machine Learning. 01 2006.

[36] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, 2011.

[37] Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. Omid, Reloaded: Scalable and Highly-Available Transaction Processing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 167–180, Santa Clara, CA, February 2017. USENIX Association.

[38] Apache Cassandra. Using lightweight transactions. https://docs.datastax.com/en/cassandra-oss/3.0/. [Online; Accessed: 2022-06-26].

[39] Justin Levandoski, David Lomet, Mohamed Mokbel, and Kevin Zhao. Deuteronomy: Transaction Support for Cloud Data. pages 123–133, 01 2011.

[40] W. Zhou, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *Services Computing, IEEE Transactions on*, 5:1 – 1, 01 2011.

[41] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. pages 21–21, 06 2012.

[42] James Corbett, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Jeffrey Dean, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford, and Christopher Heiser. Spanner: Googles Globally-Distributed Database. *ACM Transactions on Computer Systems*, 31:1–22, 08 2013.

[43] Daniel Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. Omid: Lock-free transactional support for distributed data stores. pages 676–687, 03 2014.

[44] Adewole Ogunyadeka, Muhammad Younas, Hong Zhu, and A. Aldea. A Multi-key Transactions Model for NoSQL Cloud Database Systems. pages 24–27, 03 2016.

[45] Mark Brown and Sneha Gunda. Transactions and optimistic concurrency control. https://docs.microsoft.com/en-us/azure/cosmos-db/sql/database-transactions-optimistic-concurrency#optimistic-concurrency-control. [Online; Accessed: 2021-12-15].

[46] Changqing Li and Jianhua Gu. A Surfing Concurrence Transaction Model for Key-Value NoSQL Databases. *Journal of Software Engineering and Applications*, 11:467–485, 01 2018.

[47] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment*, 14:1241–1253, 03 2021.

[48] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. pages 143–154, 09 2010.

[49] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Shlens, Vinyals, Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/, 2015. Software available from tensorflow.org [Online; Accessed: 2021-12-14].

[50] The Apache Software Foundation. What is Apache Spark? https://spark.apache.org. [Online; Accessed: 2022-08-24].

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.