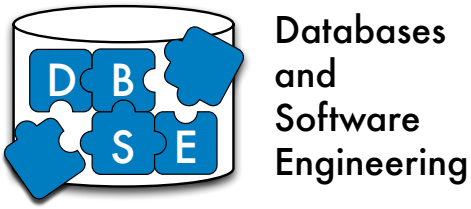
Otto-von-Guericke-Universität Magdeburg

Faculty of Computer Science



Master's Thesis

Understanding and Improving Deep Reinforcement Learning for Data Partitioning

Author:

Abdullah Al Zubaer

April 5th, 2021

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Department of Databases and Software Engineering

MSc. Gabriel Campero Durand
Department of Databases and Software Enginering

Zubaer, Abdullah Al:
Understanding and Improving Deep Reinforcement Learning for Data Partitioning
Master's Thesis, Otto-von-Guericke-Universität Magdeburg, 2021.

Abstract

Being able to optimally partition a database, for disk storage or for distributed uses, is one of the most significant physical design aspects for improving the performance of query processing on a database system. Unfortunately, finding the optimal partitions is an NP-hard problem, so instead of computing the actual optima heuristic methods are commonly adopted. Even though there exist several approaches for data partitioning, most methods cannot guarantee optimality.

In recent times there has been ongoing research for finding the optimal partitioning by using various methods, including methods based on machine learning techniques. Other than classical machine learning approaches, deep learning methods coupled with reinforcement learning, i.e., Deep Reinforcement Learning (DRL), for finding a good partitioning strategy have been taken into consideration by researchers.

In this thesis, we focus on applying DRL methods for horizontal partitioning, with a focus on using the TPC-H dataset and a distributed file system storage based on the Parquet file format. We formulate the problem of data partitioning in a reinforcement learning framework. We present a working environment suitable for applying DRL agents, with options tailored towards improving its efficiency, such as caching. We study possible gains from our approach by using two DRL agents, a basic one (DQN) and one that is augmented with memory (R2D2). Our model implementations are based on the Deepmind ACME framework, and for both cases we adopt a model improvement by introducing an action mask, which we have integrated with the architecture of the two agents and their training processes.

We report competitive results compared to baseline non-partitioned processing with overall speed-ups of around 2x-5x (when running 6 steps), but final time speedups in the range of 1.7 for the resulting partition. We also find little difference between R2D2 and DQN for our application. We find that although the loss for agents seems to be converging, agents do not converge to the most competitive solution observed during training, which suggests a need for further training and tuning. We also report that in the absence of information about the queries the agents do not generalize well to partitioning for unseen queries. To conclude we highlight areas for future work such as determining the optima with dynamic programming, evaluating on larger databases, and improvements towards generalization.

Acknowledgements

First and foremost, I would like to express my most heartfelt gratitude to my supervisor M.Sc. *Gabriel Campero Durand* for his patience and support from the very beginning till the end of this thesis. Without his continuous guidance, cooperation, and encouragement this thesis would not be possible to conclude. It was a pleasure and honor for me to work under his kind guidance. I am and I will always be grateful for everything towards him.

Furthermore, I would like to express my heartful appreciation to Prof. Dr. rer. nat. habil. Gunter Saake for allowing me to write my thesis in his department.

I would also like to acknowledge the supports and love that I have received from my family members especially my mother, from the beginning of the thesis till the end and in every aspects of life.

Finally, but most importantly, I would like to express my deep thankfulness for Fatima's constant patience, love, and encouragement in this difficult journey of this thesis and in life.

Declaration of Academic Integrity

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Signature	Place, Date

Contents

List of Figures **xii**

List of Tables **xiii**

List of Algorithms **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contribution . . . 4
1.3 Thesis structure . . . 4

2 Background **7**
2.1 Partitioning methods in database systems . . . 7
2.1.1 Horizontal partitioning . . . 8
2.1.2 Vertical partitioning . . . 10
2.1.3 Hybrid partitioning . . . 12
2.2 Related work on data partitioning . . . 12
2.2.1 Approaches using reinforcement learning . . . 13
2.2.1.1 Learning a partitioning advisor for cloud databases . 13
2.2.1.2 Qd-tree: learning data layouts for big data analytics 16
2.3 An introduction to DRL and methods of interests . . . 20
2.3.1 Reinforcement learning . . . 21
2.3.1.1 Introduction to reinforcement learning . . . 21
2.3.1.2 Markov Decision Processes . . . 23
2.3.1.3 Policy and value function . . . 25
2.3.1.4 Optimal policy and optimal value function . . . 26
2.3.1.5 Methods for solving finite MDPs . . . 29
2.3.2 Deep reinforcement learning . . . 32
2.3.3 Agents . . . 33
2.3.3.1 DQN . . . 33
2.3.3.2 R2D2 . . . 36
2.4 Summary . . . 39

3 Prototype design and research question **41**
3.1 Research questions . . . 41
3.2 Prototype design . . . 42
3.2.1 Environment . . . 42
3.2.1.1 Architecture . . . 42

3.2.1.2 Observation space . . . 43
3.2.1.3 Action Space . . . 44
3.2.1.4 Reward . . . 49
3.2.1.5 Stopping condition . . . 49
3.2.1.6 Dynamics . . . 50
3.2.2 Caching . . . 50
3.2.3 Model improvements . . . 52
3.3 Summary . . . 58

4 Experimental setup **59**
4.1 Dataset and workload . . . 60
4.2 File format . . . 62
4.3 Reward calculation . . . 62
4.4 Hyperparameters for DQN and R2D2 . . . 62
4.5 Train and test set . . . 63
4.6 Experimental Environment . . . 63
4.7 Summary . . . 64

5 Evaluation and Results **67**
5.1 Over-fitting to training examples . . . 67
5.2 Generalizing basic discrete DRL agent (DQN) . . . 78
5.3 Generalizing memory based discrete DRL agent (R2D2) . . . 85
5.4 Summary . . . 91

6 Conclusion and Future work **93**
6.1 Conclusion . . . 93
6.2 Future work . . . 94

Bibliography **97**

List of Figures

2.1 Hybrid partitioning [zV11] 12

2.2 Overview of the system design proposed by Yang et al. [YCW+20] . . 17

2.3 Example of a qd-tree [YCW+20] 17

2.4 Association between RL and other branches of artificial intelligence [Li18]. 22

2.5 RL and other fields of study [Sil15]. 22

2.6 Interaction of an agent and the environment in MDP. [SB18] 24

2.7 Approaches to solve (state/state-action) value function adapted and modified from [DHYD20]. 32

3.1 Observation space. 43

3.2 Observation space before agent chooses any column for partitioning the TPC-H data 45

3.3 Observation space after agent chooses *c_acctbal* column for partitioning the *customer* table 46

3.4 Steps per second for DQN and R2D2 w.r.t iteration during testing. . 53

3.5 Steps per second for DQN and R2D2 w.r.t iteration during training (using log scale for y axis) 54

4.1 TPC-H schema 60

5.1 Cumulative episodic return and runtime for workload execution per iteration for DQN (training dataset). 69

5.2 Cumulative episodic return and final runtime for workload execution per iteration for DQN (test dataset). 70

5.3 Loss for DQN agent. 72

5.4 Final runtime gain ratio for DQN w.r.t baseline. 73

5.5 Cumulative episodic return and runtime for workload execution against iteration for R2D2 (training dataset). 75

5.6 Cumulative episodic return and final runtime for workload execution per iteration. for R2D2 (test dataset). 76

5.7 Loss for R2D2 agent. 77

5.8 Final runtime gain ratio for R2D2 w.r.t baseline. 79

5.9 DQN generalization results for train data w.r.t episodic return and final runtime for workload execution per iteration. 81

5.10 DQN generalization results for train data w.r.t final runtime gain. . . 82

5.11 DQN generalization results for validation data. 83

5.12 DQN generalization results for test data. 84

5.13 R2D2 generalization results for train data w.r.t episodic return and final runtime. 87

5.14 R2D2 generalization results for train data w.r.t final runtime gain. . . 88

5.15 R2D2 generalization results for validation data. 89

5.16 R2D2 generalization results for test data. 90

List of Tables

2.1 Example Database [zV11]. 9
2.2 Primary horizontal partition of relation PROJ [zV11]. 10
2.3 Derived horizontal partition of relation EMP [zV11]. 10
2.4 Vertical partition of relation PROJ [zV11]. 11

3.1 Actions available for the agent. 47

4.1 Common hyperparameters in DQN and R2D2 [HSA+20]. 65
4.2 Hyperparameter for DQN [HSA+20]. 65
4.3 Hyperparameters for R2D2 [HSA+20]. 66

5.1 Episode return and final runtime for DQN. 68
5.2 Episode return and runtime for R2D2. 74
5.3 Summary comparison of **RQ1** for DQN and R2D2. 78
5.4 DQN Generalization statistics. 85
5.5 R2D2 Generalization statistics. 86

List of Algorithms

2.1 Offline Training [HBR20] 15
2.2 Greedy construction of qd-tree [YCW+20] 18
2.3 Estimating the state-value using TD(0) method [DHYD20] 30
2.4 Q-learning (off-policy TD control) [DHYD20] 31
2.5 Deep Q-learning with Experience Replay [MKS+15] 36

3.1 Pseudo-code of our approach. 51
3.2 Pseudo-code of basic DQN training 56
3.3 Pseudo-code of our usage of the legal action mask in DQN training . . 56
3.4 Pseudo-code of basic R2D2 training 57
3.5 Pseudo-code of our usage of the legal action mask in R2D2 training . . 57

1. Introduction

1.1 Motivation

Efficiently partitioning a database is one of the most significant decisions that database administrators have to make regarding the physical design of the database (i.e., when we deploy the database in the real world), since the query processing runtime heavily depends on the characteristics of the partitioned data. Optimal partitioning of the database will lead to a significant improvement in performance w.r.t query processing time since it allows the processing engine to only access the data that are required for a given workload. In addition partitions can be used to help compression and hence reduce the data used and query processing costs even more.

Unfortunately, finding the optimal partitioning is an NP-hard problem, and therefore, heuristic methods are commonly adopted for data partitioning [SW85]. There have been several works related to data partitioning previously, for example, by using greedy methods [ANY04], approximation methods [GKP+10], or others [RZML02]. Apart from methods related to partitioning only, different techniques has been proposed to automate tuning of the database system as well next to partitioning[CN07, PAA+17].

The approaches mentioned earlier for data partitioning cannot ensure optimal partitioning, since, as stated previously, data partitioning is an NP-hard problem and therefore researchers have had to come up with heuristic algorithms to solve the data partitioning problem.

Recently, researchers are coming up with approaches based on machine learning to solve the problem of partitioning data efficiently [DC04, DD11, RMAHAFPC14, HR15]. Within this context, deep learning methods have also been applied to address more general database management system (DBMS) problems[PAA+17]. Additionally, there has been proposed work with deep reinforcement learning for data partitioning [DPP+18, DPP+19, HBR20, YCW+20] and for tuning DBMS [SSD18].

Machine learning methods can be subdivided into three categories, mainly supervised learning, unsupervised learning, and reinforcement learning. Reinforcement

learning (RL) is a process of learning from the experience of an agent. Where the agent acts on an environment, and in return, the environment provides a reward signal evaluating the action chosen by the agent (which is explained in detail in Section 2.3.1 on page 21). Whereas in supervised and unsupervised learning methods, the learning process occurs based on the datasets provided and using the knowledge given by the dataset [SB18].

To perform data partitioning more systematically and to learn from experience, RL methods are considered by some researchers to be suitable. Using this approach the agent will partition the database and evaluate the partitioned results w.r.t the workload execution time, and will periodically improve the partition based on the reward signal, which will represent the quality of the action taken by the agent, from the environment.

Unfortunately, the traditional RL method mentioned in Section 2.3.1.5 on page 29 for approaching reinforcement learning problems is not applicable for solving real-world problems, since in the real world the environment is exceedingly more complex and the number of possible actions is considerably large. To overcome this shortcoming of traditional solving methods for RL problems, there have been proposed approaches, for example, [MKS+13, MKS+15], of using deep learning [LBH15] methods with RL methods. This new paradigm of this learning method is called Deep Reinforcement Learning (DRL).

Compared to heuristic methods, deep reinforcement learning requires a large amount of training to show reasonable results, and this training can be difficult due to the influence of many factors such as hyper-parameters or design of reward functions. In spite of such well-known limitations, deep reinforcement learning has some advantages for its application to problems like partitioning:

- It can be expected to generalize to new problems, thanks to the rich representation learning capabilities, the possibility of transfer learning, and also a growing number of models being developed by the community. For example, agents can learn problems that use images as inputs, or that have complex memory-related requirements.
- Models can be trained on relatively raw data and problems that are underspecified (e.g. actual execution times instead of using cost models).
- Models can be expected to become better with experience. In fact, this approach provides a well-defined framework for model evolution and development.
- Models can be pre-trained from demonstration to perform better from the beginning.
- Finally, with deep reinforcement learning pre-trained models can be used to make predictions at runtime in a way that is O(1), contrasted to methods that require to do search among candidate partitions at this stage.

In this thesis, we study the task of partitioning a database horizontally by applying Deep Reinforcement Learning (DRL). There has been a more limited amount of

work related to horizontal partitioning of a database compared to other partitioning methods. Even though there are published works related to data partitioning using deep reinforcement learning, the lack of open-source work, and the lack of using state-of-the-art DRL agents from standard libraries, makes it difficult for other researchers to follow up the current works. Furthermore, partitioning a database is a complex task, due to the different combination of approaches that can be applied to partition the database, and the different database scenarios (e.g., distributed vs. local, distributed-file-system-based vs. more traditional), leading it to be a difficult problem to study and consequently, to find the optimal partitioning. Additionally, partitioning the database optimally is highly related to the given workload, requiring different partitioning methods for different workloads, therefore it is not reasonable to determine the partitioning strategy beforehand.

We will address the problem of partitioning a database optimally w.r.t workload execution time using DRL methods. We introduce a discrete environment for partitioning a database, where the DRL agent will be interacting, called Data Partitioning Environment (DPE), that focuses on horizontal partitioning. Our environment uses Spark [ZCF+10]. As mentioned previously, we have scoped our work to horizontal partitioning exclusively, but our environment, DPE, would work for the other two approaches: vertical partitioning and hybrid partitioning. Our DPE is a single environment where we can perform data partitioning using DRL agents.

We will be focusing on a representative Spark [ZCF+10] use case with TPC-H[1] data and we will test DQN [MKS+13, MKS+15] and R2D2 [KOQ+18] DRL agents from the standard DRL framework, ACME [HSA+20]. We have adopted the DQN agent, as our first DRL agent, for our study as a representative of classical DRL methods which is appropriate for our discrete environment and actions. Additionally, we have worked with the R2D2 agent as our second DRL agent, which is a memory-based DRL agent using LSTM, unlike DQN. We believe solving the data partitioning problem will require memory of the partitions performed by the agent before for better performance in the later period, w.r.t workload execution runtime. This relevance of this model could increase with the complexity of the partitioning choices offered by our environment.

We consider our study to be relevant for developing and strengthening solutions for data partitioning based on experience. We contribute with an open source implementation and the use of standard agents from the literature.

In existing research for applying deep reinforcement learning to data management tasks[Dur], authors have highlighted problems that require better understanding, such as application-side contribution, impact of problem framing, role of model selection, and benefits from training configurations. With our research, we intend to contribute to a better understanding of application-side contributions by standardizing the problem framing on a Spark-based system design, and to further the understanding of model selection.

In our next section, we provide our major contributions in this thesis.

[1] http://www.tpc.org/tpch/

1.2 Contribution

Our contributions in this work are listed below:

1. We provide an environment for data partitioning, that can be used by others for partitioning any database using deep reinforcement learning agents given the workload, based heavily on previous work [HBR20, DPP+18, DPP+19]. Our environment design considers caching of query runtime, to improve performance, re-use of partitions, and an action mask to further guide the agents. In our environment we have used TPC-H[1] as the benchmark dataset and the workloads for our study. Furthermore we presented Spark [ZCF+10] use case with TPC-H[1] dataset for data partitioning using Parquet[2].

2. We provide early results for horizontal data partitioning using two DRL agents, DQN [MKS+13, MKS+15] representing traditional DRL agents, and memory-based agent R2D2 [KOQ+18] from the ACME framework [HSA+20] which can be used as baselines for horizontal partitioning and can be extended to other partitioning tasks as future work. We have trained and evaluated the performance of DQN [MKS+13, MKS+15] and R2D2 [KOQ+18] for horizontally partitioning a database and worked with the actual workload execution runtime as a performance measure (rewards) for the agents.

3. We present model improvement in the DRL agent by introducing an action mask, which we integrate with the architecture of the two agents (DQN [MKS+13, MKS+15] and R2D2 [KOQ+18]), as an extension, for improving their performance.

1.3 Thesis structure

This thesis is organized as below:

- In Chapter 2 on page 7 we present the necessary background needed to follow this thesis. We introduce three primary partitioning methods for a database system. We review in detail two recent research works related to partitioning the database using deep reinforcement learning methods, which includes the work by [HBR20] that we have partially adopted in this study. We further provide a detailed background on reinforcement learning in general (scoped to our thesis). We finish this chapter by introducing deep reinforcement learning in general and introducing two deep reinforcement learning agents (DQN and R2D2) that we have worked with in our study.

- In Chapter 3 on page 41 We present our three research questions that we are going to address in the thesis. We provide a detailed picture of our prototype, which includes: observation space, action space, reward calculation, stopping condition, and dynamics. Furthermore, we provide a caching strategy that we adopted for increasing the performance of our experiments and the pseudocode of our design of the environment. We end this chapter by proposing an action mask as an improvement of the DRL agents, DQN and R2D2.

[2]http://parquet.incubator.apache.org/documentation/latest/

- In Chapter 4 on page 59 we present the dataset and workload, file format, how we have calculated the reward for the agents, hyperparameters for DQN and R2D2, and our train and test strategy we have used in our experiments, along with hardware and software configurations that we have adopted for running our experiments.
- In Chapter 5 on page 67 we present the results of the experiments we have performed using DQN and R2D2 agents w.r.t to our research questions. We further present our hypothesis and discussion of the results.
- In Chapter 6 on page 93 we present the conclusions we have derived from our study of the research questions in this thesis and potential directions for future works.

2. Background

In this chapter, we are going to present the necessary background required to understand and follow our work in this thesis.

This chapter is composed in the following way:

- In Section 2.1 we present an overview of the primary partitioning methods for a database: horizontal, vertical, and hybrid partitioning.
- In Section 2.2 on page 12 we discuss in detail two recent works related to data partitioning using deep reinforcement learning, which includes the work by [HBR20] that we have partially adopted in this study.
- In Section 2.3 on page 20 we provide a background on reinforcement learning and deep reinforcement learning, we conclude this section by presenting two deep reinforcement learning agents that we have adopted in our work, R2D2 and DQN.

2.1 Partitioning methods in database systems

Partitioning a database optimally is a significant aspect of the physical design of the overall system. It intends to help us to only use the data that we require for a given workload, leading to increased performance w.r.t query processing time. Data partitioning also allows queries to execute in parallel [zV11], as different threads or processes can work with different partitions.

Partitioning is a difficult problem since the combination of allowed partitions for a database tends to be relatively high, therefore it is essential to figure out the optimal way to partition a database. Furthermore, to decide which partitioning method to use or how to partition the data is not reasonable to know in advance since partitioning a database is highly related to the given workload. Leading to different workloads requiring different partitioning methods. In this section, we present the

three primary forms of partitioning available for a given database (where we focus on the relational database only).

A relational table can be partitioned horizontally, vertically, and in a hybrid way (when both of them are used to partition the relational table).

Before proceeding to the partitioning methods, we have to be aware of the three properties that require to be fulfilled by the partitioned table, and below, we mention them [zV11].

1. Completeness: When an instance of a relation, R, is partitioned into different fragments, $F_R = \{R_1, R_2, \ldots, R_n\}$, the tuples that are present in the R must also be present in F_R, which means no data must be lost during the partitioning process.

2. Reconstructability: The partitioned $F_R = \{R_1, R_2, \ldots, R_n\}$ that are created after partitioning a relation R, we must be able to combine them with a relational function $\bigtriangledown$ to form the original relation from the partitions.

$$R = \bigtriangledown R_i, \quad R_i \forall \in F_R$$

3. Disjointness: The disjointness property is different when we partition a relational table R by horizontal and vertical partitioning methods. For horizontal partitioning, when a relational table R is partitioned horizontally into $F_R = \{R_1, R_2, \ldots, R_n\}$, then tuple T_i presents in any one of the partitioned table cannot be present in any other partitioned table, i.e. if $T_i \in R_j$ then T_i cannot be present in any other partitioned table $R_k (j \neq k)$. For vertical partitioning we cannot assert the same rule for disjointness, since in vertical partitioning the primary key attributes should be copied to all the fragments. For vertical partitioning disjointness property is for the attributes that are not primary key in the relational table.

Following, we discuss three primary kinds of partitioning methods for partitioning a relational database, including examples.

2.1.1 Horizontal partitioning

In horizontal partitioning, as the name suggests, relational tables are partitioned horizontally into sets of tuples. Horizontal partitioning can be categorized into two classes, *primary horizontal partitioning* and *derived horizontal partitioning*. In primary horizontal partitioning, the relational table is partitioned by predicates which are specified on the table itself. Whereas in derived horizontal partitioning, the partitioning on a relational table is produced based on the constraints specified on a different relational table [zV11].

Formally, the primary horizontal partitioning can be defined as below:

$$R_i = \sigma_{F_i}(R), \quad 1 \leq i \leq w$$

where R is the relation, F_i is the predicate which is used to partitioned the relation R into fragments R_i [zV11].

And derived horizontal partitioning can be defined as below:

$$R_i = R \ltimes S_i, \quad 1 \leq i \leq w$$

Where w is maximum number of partition that can be on the relation R, $S_i = \sigma_{F_i}(S)$, predicate F_i is defined on relation S only [zV11].

Table 2.1: Example Database [zV11].

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	J. Maintenance	310000	Paris

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

For example [zV11], the following primary horizontal partitioning of the example database in Table 2.1

$$PROJ_1 = \sigma_{\text{LOC=“Montreal”}}(PROJ)$$
$$PROJ_2 = \sigma_{\text{LOC=“New York”}}(PROJ)$$
$$PROJ_3 = \sigma_{\text{LOC=“Paris”}}(PROJ)$$

will partition the *PROJ* table into the following fragments shown in Table 2.2 on the next page.

Next, we will provide an example [zV11] of derived horizontal partitioning of the example database in Table 2.1.

Table 2.2: Primary horizontal partition of relation PROJ [zV11].

PROJ_1

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

PROJ_2

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York

PROJ_1

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

$$EMP_1 = EMP \ltimes PAY_1$$
$$EMP_2 = EMP \ltimes PAY_2$$

Where

$$PAY_1 = \sigma_{SAL \leq 30000}(PAY)$$
$$PAY_2 = \sigma_{SAL > 30000}(PAY)$$

The result of the above partitioning is given below in Table 2.3.

Table 2.3: Derived horizontal partition of relation EMP [zV11].

EMP_1

ENO	ENAME	TITLE
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E7	R. Davis	Mech. Eng.

EMP_2

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E8	J. Jones	Syst. Anal.

2.1.2 Vertical partitioning

In vertical partitioning, as the name suggests, the relational table is partitioned vertically. One distinct feature of vertical partitioning is every fragment of the partition should have the primary key attribute present in it (which is required for reconstructing the original relation from the partitions). In vertical partitioning,

the number of partition available is approximately equal to the Bell number, $B(m)$, where m is the number of attributes, if a relation has 10 attributes, then the relation can be vertically partitioned in approximately more than 115000 ways [HN79].

Due to the complex nature of partitioning a relation vertically, heuristic methods can be applied. In the [zV11] the authors pointed out two different approaches, 1) Grouping and 2) Splitting, and focused on the second approach. In brief, first, the attributes in the relation need to be clustered based on which attributes are related to which attributes. Then, the bond energy algorithm (BEA) [MJSW72, MDMS69] is used for producing clusters of attributes, in form of a matrix (*"clustered affinity matrix (CA)"* [zV11]), that are closely related to each other. Then the relation can be partitioned using the partitioning algorithm presented in [zV11] on page 111 by using the CA.

The following vertical partitioning of the example [zV11] database's table *PROJ* in Table 2.1 on page 9

$$PROJ_1 = \pi_{PNO,PNAME,LOC}(PROJ)$$
$$PROJ_2 = \pi_{PNO,BUDGET}(PROJ)$$

will partition the PROJ table into the following vertical fragments as shown in the table Table 2.4.

Table 2.4: Vertical partition of relation PROJ [zV11].

$PROJ_1$	
PNO	BUDGET
P1	150000
P1	135000
P3	250000
P4	310000

$PROJ_2$		
PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris

2.1.3 Hybrid partitioning

In hybrid partitioning, a relation R can be partitioned horizontally followed by vertical partitioning or vertically followed by horizontal partitioning. Due to the hybrid nature of partitioning, we get a tree-like formation as shown in Figure 2.1 from [zV11]. Where the relation R is first partitioned horizontally, which produced relation R_1 and R_2, then the partitioned relation is vertically partitioned to producing $R_{11}, R_{12}, R_{21}, R_{22}$ and $R_{2,3}$. Furthermore, the depth of the tree, i.e., number of partitions, is bounded by, for horizontal partitioning when each partition contains one tuple, and for vertical partitioning when there is one attribute in each partition [zV11].

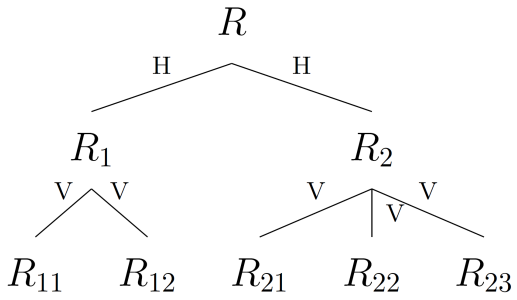


Figure 2.1: Hybrid partitioning [zV11]

In this section we have given an overview of the three primary partitioning methods, in the subsequent section, we will discuss two works based on data partitioning using deep reinforcement learning techniques, with a focus on horizontal partitioning.

2.2 Related work on data partitioning

Reinforcement learning (RL) is a paradigm of learning methods, which is a subset of machine learning. Typically, machine learning can be subdivided into three categories, supervised learning, unsupervised learning, and reinforcement learning. RL is a method of learning from experience where an agent interacts with a given system to gather its own training data, which is a contrast with supervised learning and unsupervised learning methods, where learning occurs based on a given dataset by utilizing the information provided by the dataset [SB18].

Optimal data partitioning is a complex problem to be solved due to the large possible ways to partition a given database and to come up with an optimally partitioned database. In contrast to other machine learning methods, in the domain of reinforcement learning methods, an agent needs to partition a database and then evaluate the performance improvement w.r.t query processing time continuously. The agent should learn from experience which partitioning approach is suitable to achieve the maximum performance, in other words, to find the optimal partitioning approach. Due to the nature of the reinforcement learning approach, i.e., learning from experience, we have found it suitable to work within this thesis.

Moreover, the data partitioning problem that we are addressing in this thesis is a complex one, and following the recent success in deep reinforcement learning

(DRL) field for games [MKS+13, MKS+15] where the authors proposed deep learning [LBH15] method to solve the RL problem, we adopted deep reinforcement learning techniques in our approach. DRL methods combine reinforcement learning and deep learning to overcome the limitation (explained in more detail in Section 2.3.1.2 on page 23) of traditional methods to solve RL problems.

2.2.1 Approaches using reinforcement learning

In this section, we focus on two works that are related to partitioning the database by applying deep reinforcement learning techniques. The rest of this section will summarize their work and we will point out the research gaps in their work, which motivate our own research.

The first work that we present by [HBR20], we have *partially* adopted in our work w.r.t design and dynamics of the environment, and certain techniques to accelerate the course of training.

2.2.1.1 Learning a partitioning advisor for cloud databases

In this paper, [HBR20], the authors have focused on finding the optimal horizontal partitioning methods for a given database for online analytical processing-style (OLAP-style) workload using deep reinforcement learning (DRL) method. We will briefly review their work in this section.

Their motivation behind the paper is that they argued, even though several Database Management System (DBMS) is providing solution regarding OLAP-style workloads, for example, Amazon Web Services and Microsoft Azure, but there is still need of human intervention, for example, for choosing partitioning approaches to horizontally partition a table. They also argued that even though there exist other methods for data partitioning, they are not suitable for cloud databases due to the inaccuracy of the cost model they use, so instead they propose the need for an experience-based methods like RL.

Briefly, the contributions of their work are: formulating the data partitioning problem in the DRL framework, two phases of training (online and offline), extending their work by using a group of agents.

Following, we present the essential details in their paper.

Brief overview of their approach

They have divided their training into online and offline phases, they argued that performing training directly on the database will take a considerable amount of time since a single table can take minutes to partition. In the offline phase, they have performed training on simulation and for online training, they have performed training on the real database. In the simulation, they have used a cost model based on the network to approximate the rewards for executing the queries, and in online training, they have used the real-time runtime for executing queries as a reward on a subset of the database.

Problem formulation

Here we have mentioned how the authors have formulated their problem in the framework of reinforcement learning.

- *Partitioning state*: They have considered only horizontal partitioning and replication of table T_i as the actions to be performed by the agent. They have encoded this information as, one hot encoded features, $s(T_i) = (r_i, a_{i_1}, a_{i_2}, ..., a_{i_n})$, where r_i indicates if the table T_i is replicated or partitioned, where $r_i = 1$ represents replication and $r_i = 0$ represents replication. $a_{i_1}, a_{i_2}, ..., a_{i_n}$ are the attributes.

  They have also introduced the concept of co-partitioning between two tables (derived partitions), which they claimed will minimize the exploration of partitioning that is not optimal. They have represented this information via edges between two tables.

- *Workload state*: The workload (a set of SQL queries), is modeled as the state also. The workload is modeled as the normalized frequency of each queries in the given workload, $s(Q) = (f_1, \ldots, f_m)$, where f_1 is the normalized query frequency for query-1 in the workload Q.

- *Actions*: The authors proposed three actions. At each step, the agent can choose only one of these actions. First, a table can be partitioned by using an attribute, second, a table can be replicated, and third, an edge can be activated or deactivated. Notably, a table cannot be partitioned several times by an attribute, nor can it be un-partitioned, similarly it cannot be un-replicated.

- *Rewards*: As discussed above, they have used two different rewards for two different setting of training, offline and online. For offline they have used a cost model based on the network, and for online they have used the runtime as a reward.

Following, we briefly discuss their model training procedure. They have divided the training into two phases, offline and online.

In the offline setting, training of the agent is performed with simulation of the database and where they applied their custom cost model $c_m(P, q_i)$ to estimate the cost of partitioning and executing the queries. They have defined there the cost as sum of the computation cost and the network cost for a query q_i and the partitioning methods P. This cost is used as a reward for the agent during the offline stage. In essence, the training algorithm proposed by the authors follows the following steps. As in any RL training procedure, the agent acts in an episode with a maximum number of episode $e_{\max}$, and each episodes consists of steps of maximum number $t_{\max}$. For each step in an episode, actions are chosen by the agent based on ϵ value and the action is performed on the simulation and the reward is calculated, c_m. Furthermore, the transition, which involves the current state, action, reward, and the next state, is stored and sampled from it for training the Q-network, and subsequently the target network weights are also updated with the Q-network's weight along with some modifications. The pseudo-code of offline training stage for the agent is shown in Algorithm 2.1 on the facing page.

In an online setting, training of the agent is performed directly on the real data in the cluster and the runtime for executing a query is used as a reward signal for a partitioning method. The authors have claimed that this direct approach is not

Algorithm 2.1: Offline Training [HBR20]

Randomly initialize Q-network Q_θ
Randomly initialize target network $Q_{\theta'}$
for e in 0, 1,. . . ,$e_{\max}$ **do** `// Episodes`
    Reset to state s_0
    for t in 0, 1,. . . ,$t_{\max}$ **do** `// Steps in Episodes`
        Choose $a_t = \text{argmax}_a\, Q_\theta(s_{t+1}, a)$ with probability $1 - \epsilon$, otherwise random action
        Execute action a_t (i.e., simulate what the next state s_{t+1} and partitioning P_{t+1} would be)
        Compute reward with cost model $c_m : r_t = \sum_{j=1}^{m} f_j c_m(P_{t+1}, q_j)$
        Store transition (s_t, a_t, r_t, s_{t+1}) in B
        Sample minibatch (s_t, a_t, r_t, s_{t+1}) from B
        Train Q-network with SGD and loss $\sum_{i=1}^{b}(r_i + \gamma\, \text{argmax}_{a \in A}\, Q_{\theta'}(s_{i+1}, a) - Q_\theta(s_i, a_i))^2$
        Decrease ϵ
        Update weights of target model: $\theta' = (1 - \tau)\theta' + \tau\theta$

feasible for training and they proposed some techniques that will be accelerating training period. They have proposed, *"sampling"* [HBR20], where the agent is not trained on all the data in the database, but a subset of it. Another method that they used is caching of the query execution time, the purpose mentioned is, if two states have the same partitioning then instead of performing the partitioning again and getting the runtime instead the runtime will be returned from the cache. The next method they adopted is called *"lazy repartitioning"* [HBR20], the goal is if the current partitioning that the agent perceives is not the same as the partitioning that is previously present in the database then the tables are partitioned and then the queries are executed, or else the queries are not executed and the runtime is returned from the cache maintained for the runtime. The final improvement they have proposed for online setting is to set a *"timeout"* [HBR20], which will stop the training process for a certain query if it takes more than a threshold of time and continues the training process for other queries. Their reasoning is if a certain query takes more than a threshold of time, then the partitioning that is present cannot be a good partition.

They have also proposed two optimization methods when the workload changes, below we summarize them.

They have proposed to use not a single DRL agent to recommend partitioning based on the workloads, but to train a set of DRL agents which they call *"committee of experts"* [HBR20] which will be considering different kinds of workloads (in case of workload changes). They argued that to have only one DRL agent is inefficient for recommending partitioning when the workload changes.

The next approach they suggested is based on the process of training of the DRL agent, which they called *"incremental training"* [HBR20]. They have argued that if new queries are added for training, then they only have to append the new workload

to the previous inputs and only need to train on the subset of the workload that has not been seen by the agent before, and this can be obtained because of the runtime cache for queries they maintained during the training process.

They have performed several experiments in their paper based on SSB [OOC07], TPC-DS[3], TPC-CH [FKN11] benchmarks and their corresponding workloads.

We will briefly summarize their experiments and the results. As mentioned earlier in this section, they have divided their training phase into two steps, offline and online. The first experiment they perform is based on offline training of the agent. They have compared their results with two heuristics for the benchmarks mentioned above, and in all the cases, the DRL agent provided better runtime for the given workloads. Their second experiment was based on TPC-CH, where they showed that online training of the agents after training them offline improves the performance regarding the runtime of executing the workloads. In their third experiment, they have shown that even when the data and workloads are changing, the DRL agent can still perform better and can find the optimal partitioning. In their fourth experiment, they have shown that the DRL agent can take advantage of the concept of exploration and exploitation, which enables them to explore more in the solution space. In their final experiment, they have shown that even when the experimental setup changes, their method is still able to find an optimal partitioning.

In this section, we have reviewed the work of [HBR20] which we adopted in our work also. Following, we are going to review the second research paper which is similarly based on data partitioning using DRL methods but by using a different approach.

2.2.1.2 Qd-tree: learning data layouts for big data analytics

In this paper [YCW+20] by Yang et al., they have proposed a data structure they call a *"query-data routing tree"* [YCW+20] (qd-tree), for data partitioning and processing queries. The qd-tree is utilized to assign records of data (*Big Data* [ECN15]) into blocks and for query processing by guiding the query to specific blocks and executing the query on appropriate blocks of data only. The tree has been developed by using greedy and DRL methods. They have argued that for analytical queries it is essential to keep the number of data blocks accessed lower to maintain high performance and even though there have been previous works related to how to arrange data inside a block however there have been fewer works regarding how the blocks should be created (i.e. how data should be partitioned) including what kind of data should be in the blocks - to increase the performance of queries by decreasing the number of blocks of data that are accessed by the queries. A lower number of data blocks accessed means higher performance in terms of I/O cost.

In the following portion of this section, we briefly summarize their work.

Figure 2.2 on the next page presents an overview of their system design. In brief, the qd-tree can be constructed either by using greedy or DRL methods and can be used for data partitioning and query processing. Using greedy methods is not an optimal way to create the qd-tree and it has some limitations. For this reason, instead of using dynamic programming which is also not feasible due to the large size of the

[3]http://www.tpc.org/tpcds/

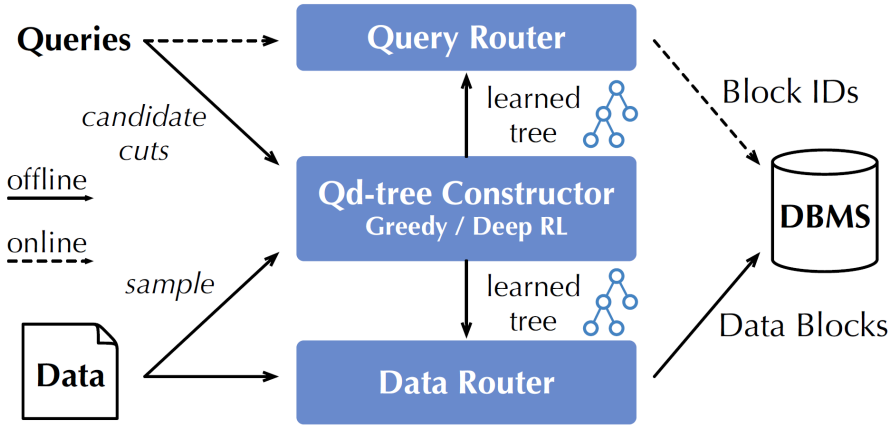


Figure 2.2: Overview of the system design proposed by Yang et al. [YCW+20]

search space, the authors resolve to use DRL for creating the tree (which works very well in extensive search spaces). The qd-tree can be used with two objectives, as mentioned by the authors. Firstly, it can be employed for creating data blocks (i.e., partitioning the data into blocks). Furthermore, it can be applied to execute queries on a block of data, where the results of the query can be expected to be found (all leaves have some semantic which guides the query execution process, ignoring blocks of data that are not relevant to the query).

For evaluation, they have performed experiments based on the TPC-H[4] benchmark dataset with two real-world workloads. Compared to other methods, they have shown that workload execution has a significant improvement in processing time and has a higher rate of data that can be skipped in the partitioned data that is based on qd-tree.

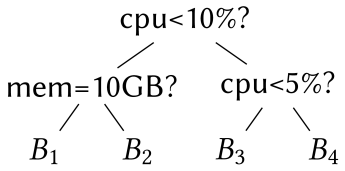


Figure 2.3: Example of a qd-tree [YCW+20]

Figure 2.3 shows a simple example of how a qd-tree would look like for a dataset with two columns. The authors have argued that each block of data in a qd-tree has the property of *"completeness"* [YCW+20] which means that the block of data has all the records that matches the predicate, and also each node has *"semantic description"* [YCW+20] which describes the kind of data that is in that node.

In the Figure 2.3 each node has a binary predicate and if the condition is fulfilled, then the data goes to the left branch of the tree, and if not, then it goes to the right branch of the tree. Here we have four leaves, which indicates four blocks of data that were created based on the predicate on each node of the tree. According to the authors, all kinds of unary predicates in the workload can be used as a

[4]http://www.tpc.org/tpch/

predicate. Moreover, each tuple is saved with an identifier known as "block ID (BID)" [YCW+20] corresponding to the block it is in, and it is later used to create the partitioning of the dataset.

The metric that they have worked with for evaluating how good the qd-tree is based on records that do not need to be touched for a given workload. The higher the quantity of data skipped for a given workload the better the tree is, and the lower the quantity of data skipped for a given workload, the worse the tree is.

Algorithm 2.2: Greedy construction of qd-tree [YCW+20]

Input: Tuple set V, min block size b, workload W, candidate cut set P
Initialization: Set $T_0 \leftarrow V, t \leftarrow 1, CanSplit \leftarrow True$
1 **while** *CanSplit* **do**
2 $CanSplit \leftarrow False$
3 **for** each node $n \in T_{t-1}$ on the last level **do**
4 **if** $n.size \geq 2b$ **then**
5 $p \leftarrow \text{argmax}_{p \in P, |n^p| \geq b, |n^{\neg p}| \geq b} C(T_{t-1} \oplus (p, n))$
6 **if** $C(T_{t-1} \oplus (p, n)) > C(T_{t-1})$ **then**
7 $T_t \leftarrow T_{t-1} \oplus (p, n)$
8 $t \leftarrow t + 1, CanSplit \leftarrow True$
9 Return T_{t-1}

As mentioned previously, the authors used greedy and DRL methods to construct the qd-tree. The pseudocode of their *greedy algorithm* is presented in 2.2. Where, T is the qd-tree, n is the node, p is the predicate and $C(T_t \oplus (p, n))$ indicates of applying the action (p, n) on the given node n on the previous qd-tree.

The root node of the qd-tree contains all records. The main idea of this algorithm is to create a tree by binary splitting of the nodes based on the optimization criterion, $C(\cdot)$, which is the total number of records that can be skipped if the split is carried out using the chosen attribute. The constraint that the author has proposed is that each newly created node (after splitting) must contain records that are larger than a threshold value ($n.size \geq 2b$) or else, the node will not be split anymore.

Here we briefly present the algorithm 2.2.

The tree is initialized with all records V, after that if the splitting condition is true, 1, then for every node in the tree, 3, the constraint mentioned before is checked, 4. If the constraint is satisfied, then the predicate p that returns the maximum $C(\cdot)$ is selected, 5. Subsequently, it is checked if the qd-tree after splitting it with (p, n) if the $C(\cdot)$ is greater than the current tree, 6, and if $C(\cdot)$ is greater than the tree is split, 7, using the predicate p on the node n then the tree is split to produce a new tree. The algorithm continues until the splitting condition is true.

In the following, we discuss the *DRL method* that they have used for building the qd-tree. The authors argued that qd-tree built using the greedy method is not optimal and to use DRL method to construct a more efficient qd-tree. They have adopted a DRL agent, which they named "WOODBLOCK" [YCW+20]. For the given dataset and the workload, the agent creates an optimized qd-tree.

The learning algorithm the agent uses for updating its parameters for the policy and value network is Proximal Policy Optimization (PPO) [SWD+17]. Both of these networks are responsible for two different tasks, the policy network tells the agent in the given state which action is better (here action means to cut on the node of the tree), and the value network tells the agent what can be the maximum long term reward that it can expect from the current state.

The agent creates several trees at each step and evaluates the quality of the tree. Based on the result of the evaluation (which is quantified by the reward signal that is measured at each step), the two network's parameters are updated by using PPO, and this continues until the termination condition is satisfied. With experience, the agent learns to create a better tree compared to the former one.

According to the authors, the termination condition is defined as follows: The leaves of the tree must contain a certain amount of records that will permit the agent to split that node further, or else the agent stops creating more nodes and the tree has been built completely. For reward calculation, $R((n, p))$ in Equation 2.2, the total reward is based on all the action/cut that has been made to construct the tree and records skipped in the node n for the workload.

$$S(n) \coloneqq \begin{cases} C(\text{n.records}) & \text{if } n \text{ is a leaf} \\ S(\text{n.left}) + S(\text{n.right}) & \text{otherwise} \end{cases} \tag{2.1}$$

$$R((n, p)) \coloneqq S(n)/(|W| \cdot |\text{n.records}|) \tag{2.2}$$

In Equation 2.2 and Equation 2.1, n is node, p is the cut corresponding to the node n, $C(\cdot)$ is the total number of record that has been skipped for a given workload, W is the workload.

Besides, the authors proposed extensions to their work regarding three aspects: 1) allowing cuts in the trees that are advanced compared to the initially proposed method of cut, 2) allowing data to replicate, leading to increased performance regarding skipping the data, 3) creating more than one tree based on the performance of the queries from the workload.

Next, we briefly discuss the experiments they have performed and the results.

For evaluation, they have adopted both logical and physical metrics. In logical metric, they have reported records accessed by the workload, and for physical metric, they have reported the end-to-end time needed for performing the operations defined by the workload. The system they have used are single and distributed Spark [ZCF+10], and a commercial DBMS. They have used TPC-H[5] and two real-world datasets, unfortunately not available to the public, (*"ErrorLog-Int"* [YCW+20] and *"ErrorLog-Ext"* [YCW+20]) and their corresponding workload. They have compared their method (qd-tree constructed by greedy and DRL agent) against three existing methods. 1) A random baseline 2) Range baseline 3) Bottom-up method [SFKX14].

[5]http://www.tpc.org/tpch/

For TPC-H[5] data, it has been shown in their experiments that qd-tree based data partitioned performed better in contrast to the other methods, both in terms of records accessed and for the runtime of executing the workload (in distributed Spark and the commercial DBMS). For the ErrorLogs dataset, qd-tree performed better both in terms of records accessed and execution time of the workload. Also, in their final experiment, the authors showed that the time required for creating good partitioning for both TPC-H and the ErrorLogs dataset is much lower compared to the Bottom-Up method of creating the partitioning.

Both the work ([HBR20] and [YCW+20]) presented in this section have utilized DRL for data partitioning using their own proposed techniques for approaching the problem. As mentioned in our motivation section (Section 1.1 on page 1), one of our goals in this thesis is to propose a single environment for data partitioning and to use advance DRL agents from the standard DRL framework available. In fact, we can refer to this as our first goal. By using a unified environment for data partitioning, the problem to solve data partitioning will be more approachable to other researchers interested in this topic. Therefore, our long-term goal would be to contribute to a single environment that is able to integrate both of the discussed approaches and to work with different types of databases and storage.

Moreover, from our review of the two recent works we find that both in [HBR20] and in [YCW+20] the authors did not work with more advanced state of the art DRL agents as available currently. Furthermore, the work of [HBR20] and in [YCW+20] are not open source, leading to the reproducibility of their work being challenging. This brings us to our second and third goals of introducing evaluations using more competitive state-of-the-art agents (in our case, R2D2), and in making the overall work open source.

Bearing these in our understanding, we aim to close these research gaps with our work in this thesis. In Chapter 3 on page 41 we present our proposed approach in detail.

In the following section, we present an introduction to reinforcement learning and deep reinforcement learning, along with the two agents that we have adopted for our study.

2.3 An introduction to DRL and methods of interests

In this section, we present the general idea behind reinforcement learning (RL) and its components, and in the following, we present deep reinforcement learning (DRL). Furthermore, we present two agents that we have adopted for our experiments in this thesis.

This section is organized in the following way:

- In Section 2.3.1 on the next page we introduce the necessary background for reinforcement learning.
- In Section 2.3.2 on page 32 we present an introduction and motivation behind deep reinforcement learning.

- In Section 2.3.3 on page 33 we present two agents (DQN [MKS+13, MKS+15] and R2D2 [KOQ+18]) which we have adopted for our experiments.

2.3.1 Reinforcement learning

Reinforcement learning (RL) is a process of learning and improving w.r.t to a task from experience over time. RL is a very general way of framing tasks that involves sequential decision making. In fact, many tasks that require to take sequential decisions can be framed as a RL problem and can be solved using RL algorithms [SB18].

In machine learning, there are three main paradigms. Generally, in supervised learning, learning is carried out based on the given dataset which is labeled, and to learn w.r.t the given task from the labeled data (for example, classification). Whereas, unsupervised learning is related to finding relations between different kinds of data that are unlabeled (for example, clustering). Finally, in RL, learning takes place by the interaction of an agent with its surroundings solely. RL does not fall into any of the above two mentioned learning methods that we are commonly familiar with. RL itself is a separate branch of machine learning on its own [SB18]. In Figure 2.4 on the following page [Li18], we can observe this in more details.

On the other hand, RL is also an intersection from different fields of studies which is shown in Figure 2.5 on the next page [Sil15].

In the next sections we will be focusing on RL and DRL, and will be discussing DRL in Section 2.3.2 on page 32.

2.3.1.1 Introduction to reinforcement learning

In this section, we present a general overview of what RL means before going into a formal description of RL.

In RL, there is an agent that interacts with its surroundings and acts accordingly by gaining experience to perform better in the given tasks. Here "accordingly" refers to a given goal, and "acts" refers to taking decision/s. Every RL agent has a goal that it wants to achieve, which means RL is goal oriented learning. Among all the fields of machine learning, RL is the closest to how humans and other animals learn. Human and other animals learn by *trial-and-error*, and RL is also learning by *trial-and-error* and this is one of the origins from where RL has emerged (from studying the psychology of how animals learn) [SB18].

We will begin informally to present some key elements in RL, and later in this section, we will present the formal definitions.

As we have discussed before, RL consists of an agent and environment. Apart from an agent and environment, RL system consists of a *policy*, a *reward signal*, a *value function* and a *model* (optionally) of the environment. A *policy* defines which action the agent will choose from a given situation. When an agent takes an action in the given state of an environment, the environment sends a *reward signal* which consists of a single scalar value called *reward*. Based on the reward the agent can "sense" if the action it has chosen was desirable or not desirable. Also, based on the reward,

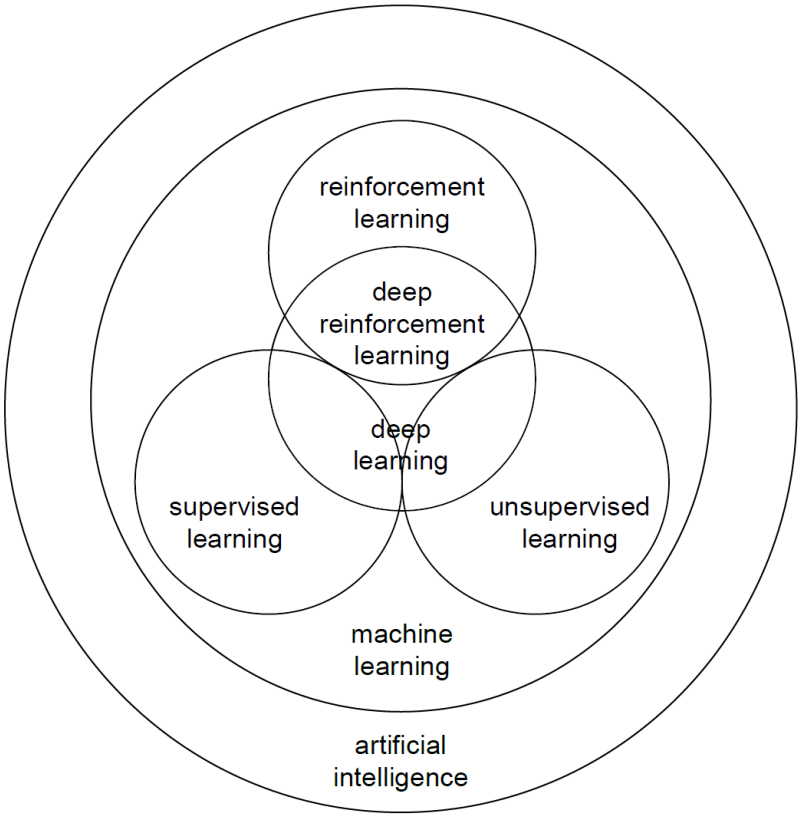


Figure 2.4: Association between RL and other branches of artificial intelligence [Li18].

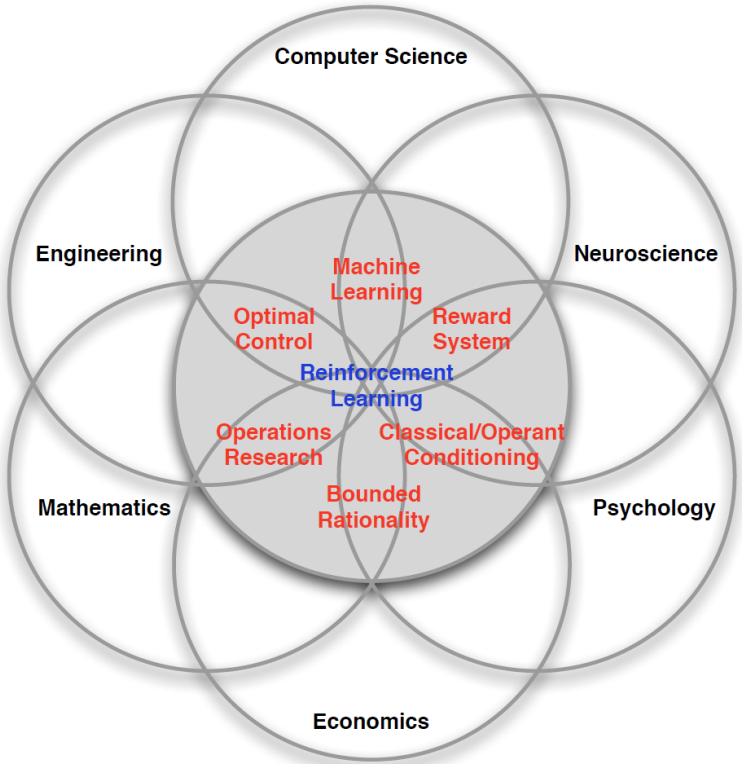


Figure 2.5: RL and other fields of study [Sil15].

the policy can be changed, for example, if a certain action chosen by the agent is not desirable, then the policy is changed in a way so that the agent tries to avoid that particular action later in the future when it comes to the same state again [SB18].

Rewards define the quality of the action taken by the agent in the given state, whereas value defines the quality of the action chosen by the agent and the state in the long run. It is natural to have a high instant reward for a given action and state, and the value of that action or state to be less desirable in the future. This leads to the fundamental goal of RL, which is to improve the value rather than the reward. Therefore, *value function* will signal the quality of the action chosen by the agent or the state the agent is in, regarding the future. For example, if a state has a high value, then the agent can expect to achieve a high cumulative reward in the future from this state, and similarly, if a state has a low value, then the agent can expect to have less reward in the future from its current state [SB18].

The fourth component of RL is a *model* of the given environment the agent is in. As the name suggests, a model of the environment will allow the agent to make reasoning about the environment in the future. For example, when a model is present for a given environment, by having a model of the environment, the model can be utilized to predict where the agent will move in the next time step and the amount of reward the agent might get in the future. Therefore, models can also be used to plan in the future before actually taking any actions from the present state. The inclusion of a model for the environment is an optional criterion, when the model is not present to the agent it is called model-free RL and when a model is present for the agent, we call it model-based RL. Both model-free and model-based RL are used to solve RL problems. When RL problems are solved using a model, it is called model-based approach, whereas when RL problems are solved without a model, it is known as model-free approach [SB18].

2.3.1.2 Markov Decision Processes

In this section, we are going to present how RL problem can be formalized mathematically in terms of Markov Decision Processes (MDP) and presents all components of an MDP.

As discussed before, any task that involves sequential decision making can be framed as RL problem and in turn, RL can be formalized mathematically in terms of MDPs. MDPs are the classical approach for any sequential decision making task. In Figure 2.6 on the next page we can see how in MDP an agent interacts with its environment [SB18].

Here in Figure 2.6 on the following page we can observe, there is an *agent* that interacts with the *environment* by taking *action* and in return the agent moves to a new *state* and the environment send a *reward* value to the agent.

Next, we will formally define the terms we have introduced before. The interaction between agent and the environment occurs in discrete time steps, $t = 0, 1, 2, 3, 4, \ldots$ and at each time stamp the the agent receives a representation of the environment called *state*, $S_t \in \mathcal{S}$, where $\mathcal{S}$ is set of states, and based on the current state the agent chooses an action, $A_t \in \mathcal{A}(s)$, where $\mathcal{A}$ is set of actions. After an action has been chosen by the agent, in the next time stamp the environment sends a reward signal

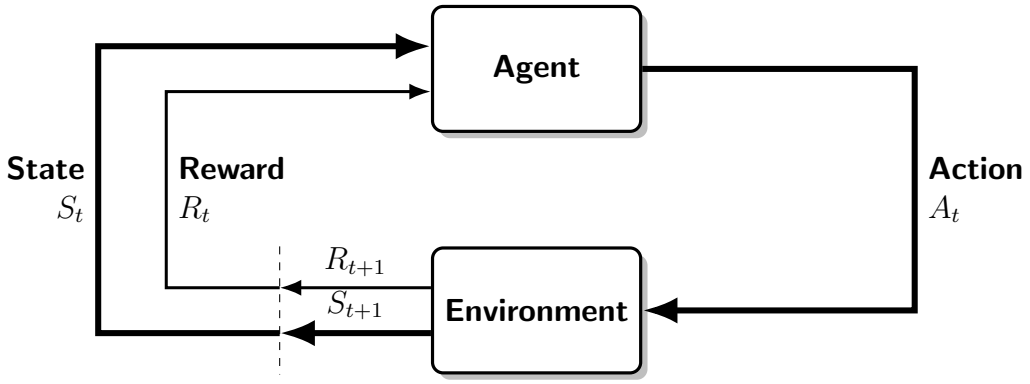


Figure 2.6: Interaction of an agent and the environment in MDP. [SB18]

to the agent based on the action chosen, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, where $\mathcal{R}$ is set of rewards, and the agent moves to a new state S_{t+1}. This interaction between an agent with its environment creates what we call a *trajectory*, $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$ [SB18].

The key property that MDP depends on is called the Markov property. The Markov property states that the current state, S_t, depends only on the previous state, S_{t-1}, and the previous action, A_{t-1}, and not on all the past states and actions. This means the last state S_{t-1} must contain all necessary knowledge regarding interaction between an agent with its environment in the past. When the previous states have the information regarding agent and its interaction with the environment from the past, then we say that the state has Markov property [SB18].

The reward signal which is sent by the environment to the agents, is one of the most important and critical aspects in RL. When the agent takes an action on a given state, the environment sends a scalar reward value, $R_t \in \mathcal{R}$, based on the action chosen by the agent. To accomplish a goal by using RL techniques, the reward signal must comply with the goal that the agent wants to achieve, since in RL the whole idea is to maximize total rewards achieved by the agent. To formalize the concept of total rewards, we will introduce the term *expected reward*. For example, for each time step t the agent receives a series of reward $R_{t+1}, R_{t+2}, R_{t+3}, R_{t+4}, \ldots$ then the total reward (*return*) the agent can achieve can be defined as below in Equation 2.3, where T is the last time step [SB18].

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + \cdots + R_T \tag{2.3}$$

In MDPs, it is convenient to consider discounted reward rather than considering the raw reward value as the return. There are several reasons for this and to explain this we will introduce the concept of *episodic task* and *continuous task*. In episodic task the interaction of an agent with its environment occurs in episodes. Each episodes ends with a termination state that is defined by the problem. Whereas, in continuous task the interaction of the agent with its environment continues without any termination state. The return function is applicable when there is a notion of episodes, on the other hand when the interaction between the agent and the given environment continues without any termination state i.e. as T approaches ∞

then the return also becomes ∞. Since the goal a RL agent is to maximize total reward, i.e. the return, formulating reward in this way is not mathematically feasible. Also, by formulating the return with discount factor allows a single formulation of the reward that can be applicable to both continuous and episodic task [SB18]. Intuitively it is very much more desirable for the agent to emphasize more on the immediate reward rather than rewards in the future [Mit97] which also relates to how human or animals gives preferences for reward that are instantaneous rather than rewards that are in the future and delayed [Sil15]. Also, since the future is not foreseeable for the given agent and due to uncertainties that future can bring, it is convenient to discount the future reward [Sil15]. Equation 2.4 shows the formulation, i.e. the definition ($\doteq$), of the return in discounted form (*discounted reward*) [SB18].

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

where:

γ = Discount rate, $0 \leq \gamma \leq 1$.
k = Time steps.

Since the goal of an agent in a RL problem is to maximize total rewards achieved by the agent, if γ is 0, then the agent is only concerned about maximizing the instant reward it gets and discarding future rewards. Whereas, as γ approaches to 1, the agent takes into consideration of maximising the future rewards, also. It can be observed from the Equation 2.4 that the fraction of reward that the agent would receive, for example in time step k, will be discounted by a factor of γ^{k-1}. Therefore the real worth of the reward if the agent received it immediately is reduced by the factor of γ^{k-1} [SB18].

The discounted reward can also be expressed recursively in the following way as shown in Equation 2.5 [SB18].

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma^1 R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.5}$$

2.3.1.3 Policy and value function

Until now, we have mentioned *policy* and *value function* without going to the details at the beginning of this section. Next, we are going to present this two key concepts in a more formal way.

A *value function* define the goodness of the given state the agent is currently present, in terms of the expected discounted return, Equation 2.4, and therefore the value function is described w.r.t to policy. A *policy* , π, is a mapping from the given state the agent is in, to the probability of choosing an action. Which means, when an

agent is following a policy π then the probability of choosing A_t in the given state S_t is $\pi(a \mid s)$, where $A_t = a$ and $S_t = s$ [SB18].

$$\pi \doteq \pi(a \mid s)$$

Whereas, *value function* is defined for the state s and for the state and action, a, pair. In an MDPs, value function $v(\pi)$ is defined for a state, s, and w.r.t a policy, π. Formally it is defined as in Equation 2.6 [SB18].

The *value function* of a given state s for the policy π, defines the expected return G_t, when the agent starts from a state s and follows the policy π [SB18].

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right], \forall s \in \mathcal{S} \tag{2.6}$$

In a similar way, the value function of an state-action pair is defined as in Equation 2.7 [SB18]. Which is known as *action-value function* or as the *Q-function* [SB18].

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right] \tag{2.7}$$

$q_\pi(\cdot)$ defines the value of taking an action a in a given state s following policy π in terms of the return G_t, i.e. if the agent chooses an action, A_t, in state, S_t, what is the expected return the agent can expect from the current time, t, and in the future by following the policy, π [SB18].

2.3.1.4 Optimal policy and optimal value function

So far, we have defined what is a policy and value function in the RL problem when it is defined as MDPs. Following, we are going to present the objective of RL, which is to find an *optimal policy*. As the name suggests, the policy, which defines the dynamics of RL, which is optimal, refers to how the agent will act in the given environment such that the maximum discounted cumulative reward (which is the return presented in Equation 2.4 on the preceding page) achieved by the agent will be maximum. To realize this concept, we will present two new terms, *optimal policy* and *optimal value function* (value function for the states and the state-action pairs) [SB18].

To specify optimal policy in an MDPs, when it is finite, a given policy, π, is better or equal to any other policy, π', if the expected return is greater or equal for all states in the environment in the other policy π' which can be mathematically written as below [SB18].

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \quad \forall s \in \mathcal{S} \tag{2.8}$$

This is known as *optimal policy* which is denoted by π_*. An π_* will have an corresponding *optimal state-value function*, $v_*(s)$ (Equation 2.9), and *optimal state-action value function*, $q_*(s, a)$ (Equation 2.10) [SB18].

The $v_*(s)$ gives the maximum expected return (Equation 2.6 on the preceding page) for any given policy in any given state.

$$v_*(s) \doteq \max_{\pi} v_\pi(s), \quad \forall s \in \mathcal{S} \tag{2.9}$$

The $q_*(s, a)$ gives the maximum expected reward (Equation 2.7 on the facing page) for any given policy in any given state-action pair (which is also known as the *Q-function*) [SB18].

$$q_*(s, a) \doteq \max_{\pi} q_\pi(s, a), \quad \forall s \in \mathcal{S}, \quad \forall a \in \mathcal{A}(s) \tag{2.10}$$

Therefore, we can say, to find the optimal policy, we have to either know the *optimal state-value function* ($v_*(s)$) or *optimal state-action value function* ($q_*(s, a)$). Both of these two functions can be solved by expressing as *Bellman optimality equation* [SB18].

The *Bellman optimality equation* for $v_*(s)$ and $q_*(s, a)$ are shown in Equation 2.11 and in Equation 2.12 [SB18].

$$v_*(s) = \max_{a} \mathbb{E}\Big[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\Big] \tag{2.11}$$

$$q_*(s, a) = \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a)\Big] \tag{2.12}$$

After we have the *Bellman optimality equation* for $v_*(s)$ and $q_*(s, a)$, solving an RL problem becomes straight forward. As we have discussed before the main objective of an RL agent is to find the optimal policy, which subsequently maximizes the total expected discounted cumulative reward, G_t [SB18].

By applying the *Bellman optimality equation* for $v_*(s)$ (Equation 2.11), the agent has to choose action in the given state that returns the maximum of *Bellman optimality equation*. Even though, at first, it does not seems to be obvious, since the policy that the agent will follow will be regarding only the current and the next state. But, due to the recursive nature of the equation, it is indeed possible to learn an optimal policy by looking ahead only one time step, since the v_* takes into consideration all future rewards [SB18].

On the other hand, if we look at the *Bellman optimality equation* for $q_*(s, a)$ (Equation 2.12), the agent has to choose an action (get the reward after moving to the next state, R_{t+1}) in the next state (from all the action available from that state) that maximizes the q value for taking an action from that state. Here we can also

observe that the agent is not required to have any knowledge regarding the functioning of the environment, and without having that knowledge, the agent will be able to determine the optimal policy [SB18].

Until now, we have presented and explained what is RL in general and then formally defined RL in terms of MPDs along with several fundamental concepts. The last two concepts, *optimal state-value function* ($v_*(s)$) and *optimal state-action value function* ($q_*(s, a)$) gave us the mathematical intuition regarding the goal of an agent in RL and how the goal can be accomplished.

Following, we are going to present a brief description of the methods available to solve RL problems when it is framed as finite MDPs. Later in this chapter in Section 2.3.2 on page 32 we will present a functional approximation method to approximately solve the RL problem using a function approximator, i.e., neural network.

Before proceeding to the next section of this chapter, we will briefly present the definitions of some fundamental concepts that we usually come across in RL literature:

- Stochastic policy : In a stochastic policy, the agent chooses actions based on the probability distribution of the actions, given in a state [DHYD20].

$$a = \pi(\cdot \mid s)$$

- Deterministic policy : In a deterministic policy, there is only one action that the agent can choose [DHYD20].

$$a \sim \pi(s)$$

- Model-based methods: In model-based methods, the agent builds a model of the environment first and then employ the model to solve the given RL problem [DHYD20].
- Model-free methods: In model-free methods, the agent does not learn the model explicitly but learns the optimal policy through trial and error [DHYD20].
- Value based methods: Value based methods are model-free methods, where the agent tries to learn the policy by optimizing the action value function [DHYD20].
- Policy based methods: Policy based methods are model-free methods, where the agent learns the policy directly, without using the value function, through trial and error [DHYD20].
- On-policy method: On policy method's goal is to analyze and upgrade the policy directly[Li18]. On-policy method is value-based and model-free [DHYD20].
- Off-policy method: Off policy method's goal is to improve the policy indirectly by learning the value function [Li18]. Off-policy method is value-based and model-free [DHYD20].

- Exploitation: In exploitation, the agent exploit the existing information about the environment and takes actions that provide the maximum reward [SB18].
- Exploration: In exploration, the agent takes actions in the environment that does not return highest reward or action that has not been taken before in order to explore actions that might provide higher rewards [SB18].
- Exploration-exploitation dilemma: In order to exploit the environment, the agent must explore the environment first. For exploring the environment, the agent might receive lower rewards compared to if the agent has exploited the current information of the environment [SB18].

2.3.1.5 Methods for solving finite MDPs

In this section, we are going to discuss the methods for solving finite MDPs.

Reinforcement learning problems, when framed as finite MDPs can be solved using dynamic programming (DP), Monte Carlo (MC), or temporal-difference (TD) learning methods [SB18]. In this thesis we will focus on TD learning methods and specifically the value function approximation algorithm.

Solving a RL problem can be summarized in two distinct ways [SB18].

1. Finding the $v_*(s)$ or $q_*(s,a)$ (i.e. optimal value function for state and state-action pair).
2. Finding the π_* (i.e. optimal policy).

In both cases mentioned above, the π_* in a given problem can be achieved if the $v_*(s)$ or $q_*(s,a)$ is available. If the $v_*(s)$ or $q_*(s,a)$ is available, then the agent only has to act on the environment accordingly w.r.t the $v_*(s)$ or $q_*(s,a)$, and subsequently this leads to following π_* [SB18].

$v_*(s)$ or $q_*(s,a)$, can be found by solving Bellman optimality equation which is for the value function and the action value function as shown below [SB18]:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}\Big[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\Big] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big] \end{aligned} \tag{2.13}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a)\Big] \\ &= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big] \end{aligned} \tag{2.14}$$

where $s \in \mathcal{S}, a \in \mathcal{A}(s)$ and $s' \in \mathcal{S}^+$ ($\mathcal{S}, \mathcal{A}$ and $\mathcal{S}^+$ are finite and when the problem is episodic problem, the final state is $\mathcal{S}^+$) [SB18].

The purpose of DP, MC, and TD methods is to discover the π_* based on the $v_*(s)$ or $q_*(s, a)$, and the method to obtain the π_* differentiates these three approaches. As stated above, we will only concentrate on TD learning methods since the scope of our thesis lies in the domain of Q-learning, which is an off-policy TD algorithm [SB18].

TD learning is based on the concept, MC, and DP methods. TD method does not require to possess a model that depicts the environment similar to MC methods (whereas DP needs a model that will represent the environment) and it bootstraps similar to DP (whereas MC methods need an episode to be completed for learning the optimal value function, and consequently for learning the optimal policy) [SB18].

The most simple form of TD learning method to evaluate a given policy, π, is as follows [SB18]:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{2.15}$$

where $V(\cdot)$ represents the value function involving the state, R_{t+1} is the reward received for the transition to the next state S_{t+1}, and γ is the discount factor. We can observe that in TD method, the state-value function is updated based on the next state's state-value and the reward, which in contrast to MC method, where the $V(\cdot)$ is updated after an episode is completed. Therefore, TD method does not require to wait until the episode finishes to update the value function. This is also known as *one-step TD* or *TD(0)* method [SB18].

For policy evaluation (which is also known as the *prediction* problem) using TD method, for each step in an episode, the agent chooses an action, A_t, based on the given policy, π, and the environment, *Env*, return a reward, R_{t+1}, and the next state, S_{t+1}. Based on the return and the state value for the next state, the current state-value function is updated for the given policy π. Below we present the pseudocode for TD(0) method for policy evaluation [DHYD20].

Algorithm 2.3: Estimating the state-value using TD(0) method [DHYD20]

Input: policy π
Initialization: $V(s)$ and step size $\alpha \in (0, 1]$
for each episode **do**
  Initialize S_0
  for Each step S_t in the current episode **do**
    $A_t \leftarrow \pi(S_t)$
    $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
    $V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$
  end
end

Next, for finding the optimal policy (known as the *control problem*), *Q-learning* algorithm was proposed by [Wat89]. In *Q-learning*, the π_* can be obtained by updating the state-action value function iteratively in each step of an episode and looking only

one-step ahead. The update equation for Q-learning is as follows (Equation 2.16) [SB18]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{2.16}$$

In Q-learning, the current state action value, $Q(S_t, A_t)$, is updated based on the reward, R_{t+1}, received for the current action and on the maximum Q value achievable from the next state, S_{t+1}, in order to approximate the optimal state-action value (Equation 2.12 on page 27) through adjusting the current Q value towards the bellman optimality equation of the state-action pair function. Where $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ is the target (from Bellman optimality equation defined for $q_*(s, a)$, Equation 2.12 on page 27), $R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ is the deviation, also known the TD error, from the target value, α is the learning rate, and γ is the discount factor.

Once the optimal state-action pair function has been discovered, the agent can act according to the Q function to behave optimally in a given environment [SB18]. Q-learning algorithm converges to the optimum action value function and it has been proved in [WD92].

The pseudocode for Q-learning ([Wat89]) is given in Algorithm 2.4 from [DHYD20]:

Algorithm 2.4: Q-learning (off-policy TD control) [DHYD20]

Initialization: Initialize $Q(s, a)$ for all state-action pairs and step size $\alpha \in (0, 1]$

for each episode **do**
  Initialize S_0
  for Each step S_t in the current episode **do**
    Select A_t using policy that is based on Q
    $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$
  end
end

Earlier what we have presented concerning the methods to solve a finite MDP follows the assumption that the actions and states can be represented in a tabular form [SB18]. In a real-world scenario, this assumption does not observe, as the actions or states are more complex, continuous, and extensive which ultimately make it an inconvenience to maintain a table for solving MPDs w.r.t memory space and computational means. Leading to Q-algorithm, presented in Algorithm 2.4 no longer is a feasible method for solving MDPs. Therefore, instead of finding the exact solution for the Q-learning algorithm, approximation approaches can be used to find an approximate solution for the Q-learning algorithm (also known as value function approximation) [DHYD20].

Figure 2.7 on the next page presents an overview of the methods that can be used for approximating the value function. When the actions or states are more complex

or continuous, several diverse kinds of function approximation methods can be used for approximately solving the value function [DHYD20]. Following the scope of our thesis, we will be focusing on function approximation using the non-linear method only, i.e. neural network.

In the subsequent section of this chapter, we will be introducing deep reinforcement learning (DRL), where we will discuss how to solve RL problems using neural network.

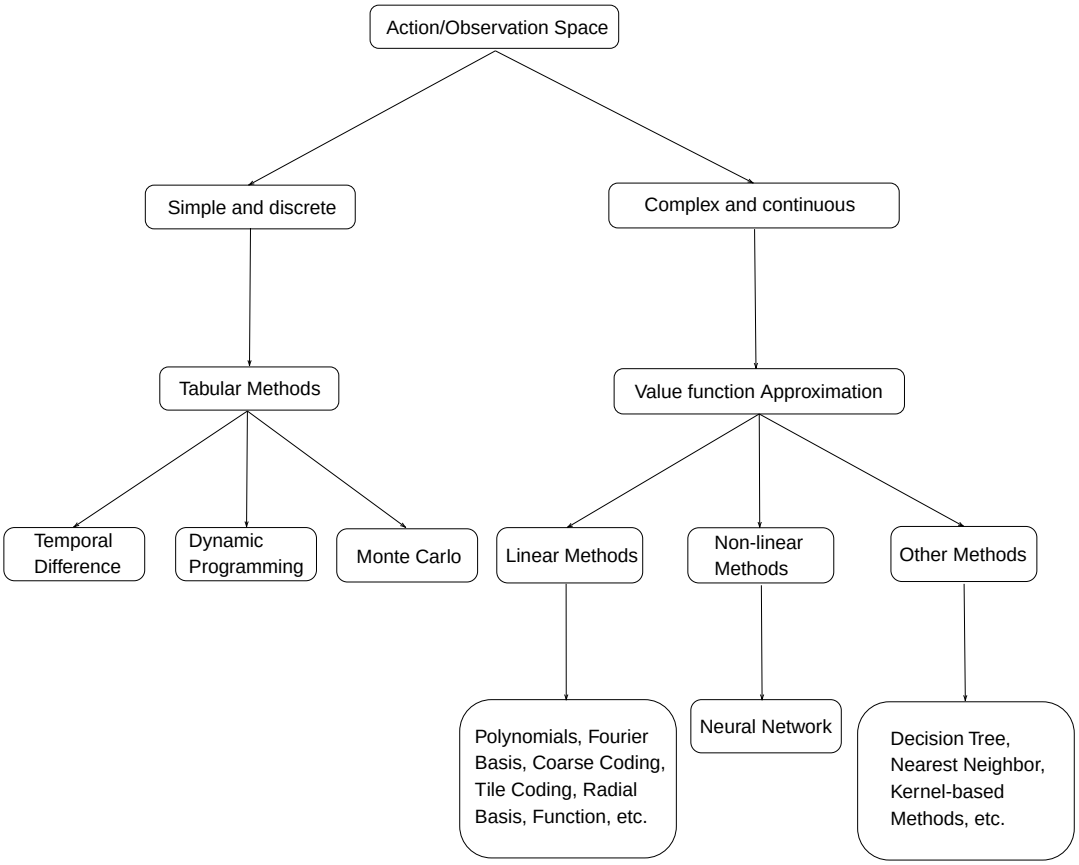


Figure 2.7: Approaches to solve (state/state-action) value function adapted and modified from [DHYD20].

2.3.2 Deep reinforcement learning

Artificial neural network (ANN) has the characteristics of approximating any function in specific situations [LLPS93, DHYD20] and is the building block for a deep neural network. One of the principal benefits of applying deep learning methods is their ability to generalize beyond training examples and to be capable to extract features from the raw input (which can be images, sound, or text). And in recent years, deep learning has seen enormous progress in the field of speech recognition, image recognition, natural language processing, language translation, and others [LBH15].

As stated in the previous Section 2.3.1.5 on page 29, solving MDPs where the action's and the state's domain are complex, the traditional method to solve Q-learning algorithm does not scale. To alleviate this matter, methods based on deep learning have been proposed and applied to solve RL problems. The first successful application of deep learning techniques for solving RL problems was proposed by Mnih et al. in their breakthrough paper *Playing Atari with Deep Reinforcement Learning* [MKS+13, MKS+15].

In the following section, we are going to present two DRL agents that we have selected for our study. We have adopted DQN[MKS+13, MKS+15] agent which is representative of traditional RL methods, and this agent is especially suitable and has been proven before to be appropriate for discrete environments and actions. Since the problem that we are trying to address consists of the environment and actions that are discrete, we have decided to use the DQN agent. We also believe that DQN will be a proper baseline before we explore other advanced DRL agents, and reasoning about the performance of the DQN agent will be a valuable insight to look into to make further improvement.

On the other hand, our second DRL agent is R2D2 [KOQ+18], which is an extension of the traditional DQN agent, with enhancement achieved by using memory in the agent's architecture. We believe that R2D2 is relevant for our study considering the problem that we are trying to address requires the memory of the partitions of the data that has been performed earlier so that the agent can utilize the past information and take more precise and intelligent decisions in the future.

We will present these two DQN agents in our following section.

2.3.3 Agents

In this section, we will provide an overview of the two DRL agents (DQN and R2D2) that we have adopted in our study.

2.3.3.1 DQN

Deep Q-network (DQN) agent was proposed by Mnih et al. in [MKS+13, MKS+15]. The authors have combined deep learning methods with RL to create a novel RL agent, i.e., the DQN agent. DQN agent is able to solve complex real world problems where the state and action domains are large and complicated and can approximate the optimal policy from high-dimensional data. They have evaluated their agent on 49 Atari 2600 games [BNVB13] and showed that the DQN agent performs better on 43 of the games compared to other RL methods, and for more than half of the games the DQN agent achieved more than 75% of the score compared to a human professional [MKS+15].

In the DQN algorithm, the authors have used 4 consecutive frames (preprocessed) from the Atari 2600 games [BNVB13] as an input to their model. In their proposed method, the authors have used the state as an input to their model with outputs as Q value (for each available action). They have argued that this procedure is more convenient concerning computation, compared to using the state and action pair as an input to the model. Otherwise, for each available action in a given state, there

needs to be a separate forward pass in the network for obtaining the Q value for the action in the given state the agent is in [MKS+15].

The network for the DQN agent consists of three convolutional layers (with 32 filters of size 8×8, 64 filters of size 4×4 and 64 filters of size 3×3, where each convolutional layer is followed by a rectifier activation function, respectively) followed by two fully connected layer (with 512 rectifier unit, and linear unit respectively). The number of linear units in the output layer is equal to the number of actions that are valid for the given game [MKS+15].

In Section 2.3.1.5 on page 29 we have discussed methods that can be used to solve finite MDPs, and the algorithm that we focused on was the Q-learning algorithm presented in 2.4 on page 31. The update equation for Q-learning presented in Equation 2.16 on page 31 shows us that to determine the optimal policy, the Q value for the current state-action pair is updated with respect to the TD error and eventually with iterative updates, Q-learning algorithm will converges to the optimal action-value function (which is the Bellman optimality equation for action-value function, $q_*(s, a)$, Equation 2.12 on page 27).

In DQN, a nonlinear function approximator, i.e., a neural network, is used to approximate action value function to estimate the optimal Q-function. Whereas, in classical RL methods, optimal action value would be estimated by iterative update with the Bellman optimality equation of the action value function as the target. As mentioned previously, in their DQN algorithm, the authors have used convolutional neural networks with fully connected layers, as the nonlinear network, to approximate the Q-function. For the target value, similar to the Bellman optimality equation for action value function in Equation 2.12 on page 27, they have employed a separate neural network with parameter θ_i^-, called the target network (y) as follows [MKS+15]:

$$y = r + \gamma max_{a'} Q(s', a'; \theta_i^-) \tag{2.17}$$

and for the estimated Q-value, they have used another neural network with parameter θ_i, called the Q-network, $Q(s, a; \theta_i)$ [MKS+15].

The loss function that is used for training the neural network in DQN is given below [MKS+15]:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right] \tag{2.18}$$

The authors have proposed two distinguished adjustments to stabilize the training procedure of a DQN agent based on the fact that when a neural network is applied to approximate the Q-function to solve RL problem, the training process grows to be unstable. This occurs due to a high correlation between:

1. the training data (the transition tuples).
2. the target and the estimated Q-values.

The correlation between the training data arises because, in RL, the agent takes action sequentially. This indicates that the next state that the agent will receive after taking an action is highly correlated with the action that was taken before by the agent.

The correlation between the target and the estimated Q-value arises if the same neural network is used for both the target and the estimation. When the neural network for target and the estimated Q-value are the same (while the Q-network is updated, the target network is also updated), it leads to the correlation between target and the estimated Q-value [MKS+15]. To mitigate these two issues, the author has proposed two techniques as discussed below [MKS+15]:

1. Experience replay [Lin93]: The experience tuple, $e_i = s, a, r, s_{t+1}$, generated by the action of the agent is stored in D, where $D = (e_1, \ldots, e_t)$ (replay memory). During training for updating the Q-value, the experience is sampled in minibatch from D uniformly, $(s, a, r, s') \sim U(D)$.

2. Target network: Instead of using the same network to estimate the Q-value and for the target value, a separate network is used for the target y_i, and the weights of the target network θ^- is updated in every C steps from the Q-network periodically.

The Deep Q-learning algorithm with experience replay along with the target network is presented in Algorithm 2.5 on the following page. Before training starts for the DQN agent, the replay memory D is filled with experiences e_t to the size of the replay memory, N (experience generated according to a uniform random policy), and the Q network and the target network are initialized with random weights of θ and θ^- respectively, where $\theta = \theta^-$ [MKS+15].

At each time step in an episode, an action is chosen randomly with a probability of ϵ or greedily from the set of available actions in the given game. After selecting an action, it is executed on the environment and the reward r_t and the next state, x_{t+1}, is received by the agent from the environment. The current and the new state are preprocessed (ϕ_t, ϕ_{t+1} respectively) and stored in D (experience replay). The target, y_j is set to the reward r_j if the episode terminate otherwise y_j is set to $r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$. Then, based on the loss of the target and the calculated Q value, stochastic gradient descent is applied to update the network parameter of the Q-network. And, at every C (the author's has used $C = 1000$) steps the network weights from the Q network (θ) is copied to the network weight of the target network (θ^-) [MKS+15].

The DQN algorithm presented in this section has been a monumental work in the field of RL, where the author has shown that the combination of deep neural networks with reinforcement learning can solve problems that have a large, complicated action and state domain. They have also shown that the DQN agent is able to score better in more than half of the 49 Atari 2600 games.

In our next section, we are going to present the second DRL agent, Recurrent Replay Distributed DQN (R2D2) [KOQ+18], which was the first agent that outperformed the human expert player in 52, out of 57, Atari games.

Algorithm 2.5: Deep Q-learning with Experience Replay [MKS+15]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, M **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for t = 1,T **do**
        With probability ϵ select a random action a_t
         otherwise select $a_t = \max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t$, a_t, x_{t+1} and prepossess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j =$
        $$\begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
        Every C steps reset $\hat{Q}$ = Q
    end
end

2.3.3.2 R2D2

Recurrent Replay Distributed DQN (R2D2) agent was proposed by Kapturowski et al. in [KOQ+18]. It is an extension of the original DQN agent [MKS+13, MKS+15], with two major advancements in the architecture, that includes:

1. Using memory based neural network, LSTM [HS97], layer in the agent's network architecture for solving partially observable Markov Decision Process (POMDP) [HS17].

2. Using prioritized experience replay [SQAS16] in a distributed manner [HQB+18].

R2D2 was the first agent that outperformed human expert players in 52 out of 57 Atari games.

In R2D2, the environment is modeled as POMDP, in contrast to modeling of the environment for DQN [MKS+13, MKS+15] agents, where the environment is modeled as Markov Decision Process (MDP). The principal difference between MDP and POMDP is that in POMDP the observation that the agent receives after taking an action on the environment is partially observable and does not contain the complete information of the current state of the environment. POMDP has been solved by applying a memory-based neural network (recurrent neural network, RNN, [LBH15]) first in [HQB+18], where the authors have used an LSTM [HS97] layer in the second last layer instead of a fully connected layer (as opposed to the DQN network

architecture) and used a single frame as a current state instead of 4 consecutive frames. We will briefly introduce the work of [HQB+18] before progressing further considering that it is necessary to present a summary (excluding the details) of the first work on memory-based neural networks and reinforcement learning.

- **Deep Recurrent Q-learning for Partially Observable MDPs** [HS17]: In this paper, the authors have proposed to apply a memory augmented neural network (LSTM [HS97]) in the DQN [MKS+13, MKS+15] architecture and introduced a new agent called *Deep Recurrent Q-Network* (DRQN). They argued that in the real world the state that the agent receives from the environment is not fully observable, in another word they are partially observable states. In the original DQN [MKS+13, MKS+15] paper, the agent was trained on four consecutive frames of the Atari games [BNVB13] (which provided complete information of the state the agent is observing) and the DQN agent was able to play and score better than a human expert for more than half of the 49 Atari 2600 games [BNVB13]. However, DQN's performance decreases when the input states are partially observable. Therefore, to investigate their hypothesis, they have introduced a memory-augmented neural network in the original DQN algorithm where they applied LSTM in the second last layer instead of the fully connected layer and modeled the Atari games in POMDP. For introducing a partial observation to their proposed approach, the authors used a single frame as input and used frames that are not fully observable, i.e., with some probability, part of the frame is hidden.

  Their proposed agent, DRQN, was able to perform equally with the traditional DQN agent in nine Atari games, despite the fact that their agent was able to see only one frame at each step.

The idea of using prioritized experience replay [SQAS16] in a distributed way was first proposed by [HQB+18]. Prioritized experience replay is an extension of the experience replay method [Lin93] that we have presented in Section 2.3.3.1 on page 33. In prioritized experience replay, the experience tuples are sampled from the replay buffer that has more priorities (i.e. informative) than other experience tuples. The main purpose behind this development is that an RL agent will be able to learn more from some experiences than others, therefore it is essential to sample the experiences with priority, from the replay buffer, for training the agent, which eventually leads to better learning for the agent [SQAS16].

This approach of prioritized experience replay was merged with distributed training in [HQB+18], which is known as Ape-X. The architecture of R2D2 is most similar to the Ape-X algorithm, therefore we will briefly present the fundamental idea of Ape-X below:

- **Distributed Prioritized Experience Replay (Ape-X)** [HQB+18]: In this paper, the authors have proposed a method to generate more data for the RL agent and, to use the prioritized experience replay method to sample from the replay buffer. In their algorithm, they have proposed to have a different process

for learning (policy) and another process for acting on the environment (generating experiences). "Actors" are responsible for creating experiences, they act on their individual copy of the environment and generate experiences that are stored in a replay buffer with preliminary priorities. Whereas, a "learner" sample from the replay buffer and update its policy including the priority of the experience tuple in the replay buffer. The network of the actor is updated periodically with the network parameters of the learners.

Ape-X was able to achieve state-of-the-art results in both discrete and continuous tasks. In one of their experiments with Ape-X DQN (which is the combination of using the Ape-X algorithm with extensions of the DQN [MKS+13, MKS+15] algorithm), they have shown that, as the number of actors increases, the expected reward also increases (in 6 Atari games [BNVB13]). They have hypothesized that this happens due to the presence of several actors responsible for generating experiences in their own way (each actor has their own unique policy), which helps to overcome the difficulty in DRL methods of not finding a policy that is a global optimum but the local optimum. This challenge can be overcome due to the presence of several actors acting on their individual copy of the environment and generating experiences, and hence the learner gets to explore more experiences than previously possible.

R2D2 has merged both of these techniques, accompanied with modifications of applying prioritized experience replay [SQAS16] in a distributed way from [HQB+18] and using memory-based neural network (LSTM) from [HS17].

Following, we are going to present how to apply prioritized experience replay in a distributed way to a memory-based neural network.

To use prioritized experience replay in a distributed way with memory-based neural networks is a challenging task because the initialization of the memory-based neural network is not straightforward since the states are partially observable in POMDP. Previously, two methods have been proposed in [HQB+18] for training the LSTM [HS97] with experience replay to work with partial observable states:

1. When sampling a sequence of experience from the replay buffer, the LSTM network is initialized to a zero state.

2. To replay a full trajectory in an episode.

In this paper (R2D2 [KOQ+18]), the authors have argued that these two methods are not suitable for training the LSTM with experience replay and therefore proposed two new techniques.

1. Stored state: Instead of initializing the RNN to zero state, the hidden states in the RNN are stored in the replay buffer, which during the training period is used to initialize the RNN i.e. with the hidden states.

2. Burn-in: The initial state of the RNN is not initialized from the full trajectory but instead by selecting a fraction in the replay sequence (after unrolling) for the initial state and the left portion of the sequence for network updates.

After evaluating the two above-mentioned methods and comparing them with the [HQB+18]'s methods, it has been found that the stored state and burn-in state provide several advantages over the previous methods for training RNN with distributed experience replay.

The architecture of R2D2 consists of 3 convolutional layers like DQN, with an LSTM layer consisting of 512 hidden units which is the input to two separate feed-forward networks (value head and advantage head) in a dueling network architecture [WSH+16].

R2D2 was evaluated on Atari-57 [BNVB13] benchmark where R2D2 achieved state-of-the-art results and in 52 out of 57 games, it performed better than the human expert. Following, R2D2 was evaluated on DMLab-30 [BLT+16] for 30 3-Dimensional games, where it also achieved state-of-the-art performance. For investigating the importance of memory augmented neural networks, the authors have evaluated their agent with a modified (with no RNN but with feed-forward neural network) R2D2 agent and have shown that the performance of the R2D2 agent with LSTM layer performed much better than without an LSTM layer.

R2D2 agent presented in this section was able to achieve the state-of-the-art result on Atari-57 benchmark using memory-based neural network and proved that memory-augmented DRL agent plays a significant role in solving RL problems.

2.4 Summary

In this section, we have presented the fundamental background that is required for this thesis to follow. We have presented an overview of the primary partitioning methods that are available in a database system, horizontal partitioning, vertical partitioning, and hybrid partitioning. Next, we have presented two works that are related to data partitioning using deep reinforcement learning, which includes the work by [HBR20] that we have partially adopted in our study. In our last section of this chapter, we have presented an introduction to reinforcement learning and deep reinforcement learning. Finally, we have presented a summary of two deep reinforcement learning agents (the first reinforcement learning agent using deep neural network, DQN, and a memory-based deep reinforcement learning agent, R2D2) that we have adopted for our experiments and evaluation.

In our next chapter, we will be presenting the research question for our work, and a comprehensive description of our prototype design for evaluating our research questions.

3. Prototype design and research question

In this chapter, we present the research questions that guide our work. We also provide a detailed picture of the prototype that we designed to evaluate our research questions.

This chapter is organized in the following way:

- In Section 3.1 we present the research questions that we are going to address in this thesis.
- In Section 3.2 on the next page we present and discuss the environment that we have created for data partitioning along with pseudocode of our proposed approach. Specifically, we address how this environment realizes the requirements of an RL framework: the actions, the reward, the state representation. Furthermore, we present the stopping conditions of the training process and the dynamics of our proposed method. Additionally, we present a caching strategy that we adopted to increase the efficiency of our experiments. We end this section with our proposed design for an action mask, which we have combined with the agent's architecture for model improvements.
- In Section 3.3 on page 58 we briefly summarize the content of this chapter.

3.1 Research questions

In order to guide our work in tackling the identified research gaps, we propose three research questions:

1. To what extent are discrete DRL models (such as DQN and R2D2) able to achieve good rewards on a partitioning use case when overfitting to the training scenarios?

2. To what extent is a traditional discrete DRL algorithms (such as DQN) able to generalize beyond training experience on the partitioning use case?

3. To what degree is a modern memory-based discrete DRL algorithm (such as R2D2) able to generalize beyond the training experience for the partitioning use case?

3.2 Prototype design

In this section, we present in detail the components of the prototype that we designed to evaluate our research questions.

3.2.1 Environment

For designing our environment, we have to consider a few aspects, bearing in mind how the problem should be modelled in an RL framework and in such a way that it can be used for solving the problem with DRL methods. Below we have given a short description of each component of our environment, and an explanation of the design choices we have made to better answer to our proposed research questions.

3.2.1.1 Architecture

For a problem to be modelled in an RL framework, we need to have three key components in the environment where the agent will be interacting, which are shortly mentioned below.

- Observation space : This defines the shape (the number of dimensions) and data types of the features of our model. It presents how the state of the environment is perceived by the DRL agent. For our case, we have a 2D matrix representing the observation space, which is explained later in this section.

- Actions : Actions define how an agent interacts with an environment. Each action has a semantic to it determining the reward and how the environment's state will change. Actions can be determined to be discrete or continuous. We work with discrete actions. For our problem, we define actions to mean the selection of a column for partitioning horizontally a given table from the schema. Furthermore, to restrict the problem, an agent is allowed to take only one action per table in an episode. On the one hand this helps to avoid hierarchical partitioning, which could be time-consuming to implement and could slow-down the training process. On the other hand, this restriction implies that our set of possible actions at the beginning of a game are all columns of all tables, but later, as the agent acts, the specific column that has been chosen by the agent (i.e. the action) to partition the table and the corresponding columns of the same table become invalid. To support this, we define at each time step a set of valid and invalid actions, from which the agent can only select from the valid ones. We explain this in detail later in this section.

- Reward : After performing an action, the agent requires a signal to understand the value of its act. This signal is the reward. We discuss in this section how the reward has been calculated in our experiments.

3.2.1.2 Observation space

Our observation space consists of a 2D matrix based on the schema of the TPC-H[6] benchmark dataset. Each column and its corresponding row represent a column from TPC-H[6]. Each dimension has a length of 61, which corresponds to the unique number of columns present in the TPC-H[6] dataset. The shape of our observation space is 61×61. We have decided to use a matrix representation for our observation space by keeping in mind that we are going to use DRL agents and that a representation able to capture the spatial relation between columns could be useful over a representation more simple like a 1D vector. Furthermore, we envision that using this representation we can include queries into our approach, with the values in the matrix, for example, indicating the percentage of queries or overall runtime that accesses a specific column.We also select this representation as a simplified approach over the case of a more expressive yet complex graph/network-based representation.

Figure 3.1 shows a 2D matrix representing our observation space, $O_{m,n}$. Where, m is the row's index and n is the column's index. Each diagonal entry of the matrix represents a column of the TPC-H[6] dataset. For example, entry $a_{1,1}$ represents column *c_acctbal*, entry $a_{2,2}$ represents *c_address* column of customer table, and so on. We have sorted each dimension of the observation matrix in alphabetical order of the column names.

Our observation space changes based on the action taken by an agent. Initially every index entry in $O_{m,n}$ is set to zero. As discussed in details in Section 3.2.1.3 on the following page, an agent is allowed to choose only actions, which is horizontally partitioning of a table by a column's attribute. For example, if an agent chooses to partition the customer table by the column *c_acctbal*'s attribute value, then the observation space changes and will have 1 in all the places representing the columns of customer table (which is the diagonal of the $O_{m,n}$ matrix). For future work we consider other values other than one can be used to better indicate the different partitions generated.

$$O_{m,n} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} & a_{1,n} \\ a_{3,1} & a_{3,2} & \cdots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n-1} & a_{m.n} \end{bmatrix}$$

Figure 3.1: Observation space.

Below we are going to present how our observation space is perceived by the agent initially before taking an action and after taking an action by the agent. We chose, as an example, the *customer* table and *c_acctbal* column to partition the *customer*

[6] http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf

table (i.e. the agent chooses to partition the *customer* table with the attribute's unique value of *c_acctbal*'s column).

In Figure 3.2 on the next page we present the observation space as it is perceived by an agent initially when no action has been performed on any table in the dataset. We can see that all entries of the observation state matrix are 0, which indicate that all tables are in their initial non-partitioned state and the agent has not performed any action (i.e. select a column from the tables for partitioning it).

When the agent chooses *c_acctbal* to partition customer table, the observation space changes, i.e., the state changes and we have 1 in all the indexes representing the customer table's column. In Figure 3.3 on page 46 we present how the observation space is perceived now by the agent after it chooses and perform an action i.e. to partition *customer* table horizontally using the attribute value of *c_acctbal*'s column.

In the following section, we are going to present the actions and the action space that we have defined for the agents.

3.2.1.3 Action Space

We have to define a set of actions that the agent can choose from, which is the domain of all possible actions, known as *Action Space*. The action space can either be discrete or continuous, in our study we have considered the action space to be discrete, considering that the agent will choose a specific column of a table for partitioning it horizontally.

Originally, when we have started to work on this thesis, we had the following actions that we wanted to consider, following the work of [HBR20]:

1. Horizontal partitioning of a table.
2. Replication of a table.

Since we have focused on experiments using a single server and not a distributed system, we decided to keep our action space limited to horizontal partitioning only, considering that replication of data mostly improves the performance in a distributed system. Our work can be easily extended for distributed systems, and this is one of our intended future work.

The action space consists of 61 discrete actions. The agent can choose an action from 0 to 60 inclusive, in every step of an episode. Each integer number represents a single column from the TPC-H[7] data. The agent can choose any column from any tables in the dataset to partition it horizontally. The action that the agent can choose from, and the column name corresponding to the action number is shown in Table 3.1 on page 47. For example, the agent can choose an action numbered 20, and the lineitem table will be partitioned using the attribute value of l_shipinstruct's column.

We have characterized the actions in our work in two ways, based on the work of [HBR20]:

[7] http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf

$$
O_{m,n} = \begin{array}{c}
\\
c_acctbal \\
c_address \\
c_comment \\
c_custkey \\
c_mktsegment \\
c_name \\
c_nationkey \\
c_phone \\
l_comment \\
\vdots \\
s_suppkey
\end{array}
\begin{array}{c}
\begin{array}{cccccccccccc}
c_acctbal & c_address & c_comment & c_custkey & c_mktsegment & c_name & c_nationkey & c_phone & l_comment & \cdots & s_suppkey
\end{array} \\
\left[\begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0
\end{array}\right]
\end{array}
$$

Figure 3.2: Observation space before agent chooses any column for partitioning the TPC-H data

$$
O_{m,n} = \begin{array}{c} \\ c_acctbal \\ c_address \\ c_comment \\ c_custkey \\ c_mktsegment \\ c_name \\ c_nationkey \\ c_phone \\ l_comment \\ \vdots \\ s_suppkey \end{array}
\begin{array}{c}
\begin{array}{ccccccccccc} c_acctbal & c_address & c_comment & c_custkey & c_mktsegment & c_name & c_nationkey & c_phone & l_comment & \cdots & s_suppkey \end{array} \\
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0
\end{bmatrix}
\end{array}
$$

Figure 3.3: Observation space after agent chooses *c_acctbal* column for partitioning the *customer* table

Table 3.1: Actions available for the agent.

Action	Column	Action	Column	Action	Column
0	c_acctbal	21	l_shipmode	42	p_partkey
1	c_address	22	l_suppkey	43	p_retailprice
2	c_comment	23	l_tax	44	p_size
3	c_custkey	24	n_comment	45	p_type
4	c_mktsegment	25	n_name	46	ps_availqty
5	c_name	26	n_nationkey	47	ps_comment
6	c_nationkey	27	n_regionkey	48	ps_partkey
7	c_phone	28	o_clerk	49	ps_suppkey
8	l_comment	29	o_comment	50	ps_supplycost
9	l_commitdate	30	o_custkey	51	r_comment
10	l_discount	31	o_orderdate	52	r_name
11	l_extendedprice	32	o_orderkey	53	r_regionkey
12	l_linenumber	33	o_orderpriority	54	s_acctbal
13	l_linestatus	34	o_orderstatus	55	s_address
14	l_orderkey	35	o_shippriority	56	s_comment
15	l_partkey	36	o_totalprice	57	s_name
16	l_quantity	37	p_brand	58	s_nationkey
17	l_receiptdate	38	p_comment	59	s_phone
18	l_returnflag	39	p_container	60	s_suppkey
19	l_shipdate	40	p_mfgr		
20	l_shipinstruct	41	p_name		

- Valid action
- Invalid action

Definition 3.1 (Valid action) *A valid action is an action where the agent chooses to partition a table that has not been partitioned before in a single episode.*

Definition 3.2 (Invalid action) *An invalid action is an action when the agent chooses to partition a table that has been partitioned before in the same episode.*

Concisely, in an episode, a table can be partitioned only once. Which means, for example, in an episode, if the agent chooses to partition *customer* table using *c_acctbal* columns, then in the same episode the agent is not allowed to choose to partition the customer table using any other columns. In brief, the agent is not allowed to partition a table more than once in a single episode.

For implementation purposes we also invalidated actions where the partition creation or overall query processing exceeded a timeout threshold during the overall training experience gathered. This was implemented in the following way for all training scenarios: at first, no assumption was made on which actions might exceed the threshold. Afterwards, every time we saw an action exceed the threshold for partition creation or query processing, we stored this action for future reference and started to consider it an invalid action for the rest of the training process.

We have implemented an **action mask** for the DRL agent as an improvement to the model. The action mask allows the agent to differentiate between valid and invalid actions, which subsequently improves the performance of the agent. Also, we have implemented the logic of action mask inside the environment which acts as a second stage verification of the actions. Details of the action mask are presented in Section 3.2.3 on page 52.

Briefly, in *an episode* the agent is allowed to choose an action that corresponds to partitioning a table using a column and by its attribute value. If the agent chooses to partition the same table with any other column or by the same column, then it is an invalid action.

In this section, in Figure 3.2 on page 45 we showed how the observation space is perceived by an agent when no action was and in Figure 3.3 on page 46 we showed how the observation space perceived by an agent after an action has been performed on the environment. We have also discussed how the observation space changes based on the actions in Section 3.2.1.2 on page 43 and how this relates to valid and invalid actions.

In our next section, we are going to present how we have calculated the reward signal for the agent.

3.2.1.4 Reward

The reward signal is the most significant and critical element of any reinforcement learning problem that requires thoughtful study. As discussed in Section 2.3.1.2 on page 23, in RL the core idea is to maximize the cumulative rewards by the agent [SB18].

The reward is usually a scalar value, given by the environment in response to an action by the agent on the environment, that signifies the "goodness" of an action taken by an RL agent. By monitoring the reward value, the agent acts accordingly on the environment. For example, if the reward value is high for a given particular action, then the agent will generally tend to choose that action in the future.

We have decided to use the following Equation 3.1 to calculate the reward signal, *r(L)*, as proposed in [SSD18] in Section 2.3 for its integrity and yet being effective.

$$r(L) = max\left(\frac{cost(\theta)}{cost(L)} - 1, 0\right) \tag{3.1}$$

Where, the cost function $cost(\theta)$ signifies the runtime for executing workloads on the non-partitioned data and cost function *cost(L)*, signifies the runtime for executing workloads on the partitioned data. The intuition behind this is : for the reward signal to be greater than 0, the runtime for executing workloads on the non-partitioned data must be higher than the runtime for executing the workload on the partitioned data. If the runtime of executing the workload on partitioned data is higher than the runtime of executing the workload on the non-partitioned data, the reward will be 0.

This means the agent will be tending to choose actions that provide lower workload execution runtime, which in turn indicates that the agent will try to partition the dataset in such a way so that the workload execution time is lower in the partitioned data compared to, when the workload is executed on the non-partitioned data.

3.2.1.5 Stopping condition

For our experiments, we have considered three stopping conditions for an episode to terminate, which is essential for the agent to know when the given episode will be over. An episode will be terminated when:

1. The agent takes an invalid action (defined in Section 3.2.1.3 on page 44).
2. The agent exceeds a given number of steps in an episode, for our experiment the number of steps is 6.
3. It takes more than a certain amount of time to perform an action by the agent and to execute the workload, for our experiment the time is 100 minutes.

When an episode terminates, the environment is reset to the initial state. Throughout our experiments, we have kept the stopping conditions fixed.

3.2.1.6 Dynamics

In this section we are going to summarize the dynamics of our environment.

When an episode begins, the agent chooses an action from a discrete set of actions, A (which ranges from 0 to 60, inclusive, Table 3.1 on page 47), based on the action mask discussed in Section 3.2.3 on page 52. The number of steps allowed in an episode is 6. If the action is invalid (according to the action mask), the episode is terminated and the agent in return receives a reward of -1 and the environment is reset to its initial state.

Whereas, if the action is valid, then it is checked if the workload execution runtime is already cached, if the runtime is cached, then the reward is calculated based on Equation 3.1 on the previous page discussed in Section 3.2.1.4 on the preceding page, without executing the action on the environment. Otherwise, we check if the file can be cached from the file cache system. Both of these caching strategies are discussed in detail in Section 3.2.2.

After the partitioning action is performed on the environment, the workload is executed on the dataset and the execution time is returned, which in turn is used to calculate the reward (Section 3.2.1.4 on the preceding page) according to the Equation 3.1 on the previous page. And, if the partitioning action takes longer than a threshold time, then the episode is terminated and the agent receives a reward of -1 and the environment is reset to its initial state.

Concerning the observation space, as discussed in Section 3.2.1.2 on page 43, the observation space i.e. the state, is represented as a 2D matrix and based on the action chosen, the observation space changes accordingly. Initially the observation space consists of all zeros and after an action is performed, based on the action, the entries on the observation space is changed to 1, which is corresponding to the the entries of the action and all the columns of that table on which the action was performed. And, when an episode terminates, the environment is set to the initial state of consisting only zeros for every index in the state matrix.

Figure 3.2 on page 45 shows how the observation space looks like before performing any action, and Figure 3.3 on page 46 shows how the observation space looks like after taking an action.

The pseudocode of our approach is presented in 3.1 on the next page.

3.2.2 Caching

For enhancing the performance of our experiments regarding experiment runtime, we have implemented two caching strategies (following the work of [HBR20]) which we are going to present in this section.

The two caching strategies we have adopted are:

1. Runtime caching.
2. File caching.

Algorithm 3.1: Pseudo-code of our approach.

Input: Dataset D, action set A, sequence of action SA, workload W, environment ENV, observation O, number of allowed steps S_t, maximum time T_{max}, runtime cache RC, file cache FC

Initialization: Set $FC \leftarrow \{\}, RC \leftarrow \{\}, non_partitioned_runtime \leftarrow R_{NP}$

```
for episode = 1, ... N do                                   // Episodes
    for steps = 1, ... S_t do                               // Steps
        Select action a_t ∈ A
        if a_t is a valid action then                       // Action mask
            SA ← a_t
            if SA ∈ RC then                                 // Runtime caching
                R_t = cached runtime
                Reward = max(RNP/R_t − 1, 0)
                ENV ← Reset
                return Reward
            else if F_t ∈ FC then                           // File caching
                F_t = file location
            Start timer ← T_timestart {
                 Perform action a_t on D, then execute W on D}
            if T_timestart < T_max then
                R_t = Execution runtime
            else                                            // Timeout ← True
                ENV ← Reset
                return Reward = -1
        else                                                // Invalid action ← True
            ENV ← Reset
            return Reward = -1
    Reward = max(RNP/R_t − 1, 0)
    ENV ← Reset
    return Reward
```

In our experiments, as mentioned in Section 3.2.1.3 on page 44, we have allowed only one action per table in an episode and we have fixed the number of actions that can be taken in an episode by an agent. During an episode, an agent can perform a certain number of actions, and each action (i.e. valid action) must be on different tables at each time. Following each action taken by the agent, we have stored two separate pieces of information regarding the runtime of executing the workload and the location of the parquet file (containing the partitioned data) created for that particular action.

We have performed this caching in two stages sequentially. In the first stage of caching, we verify if the runtime can be cached from the memory for each action or sequence of action. If the runtime does not exist in the memory, then we check if the partitioned file (based on the current action) can be cached from the memory.

In the first stage, we have saved the execution time for each partitioning action and sequence of actions (for the given workload) so that in a succeeding period during our experiment, if the agent chooses to perform the same action or sequence of action then the runtime can be read directly from the cache without executing the action on the dataset and without calculating the reward. Furthermore, considering in training DRL agents the number of the episode is very high, and in the beginning, the agent will explore the environment, there will be actions that have been performed before, hence decreasing computational time.

If the first stage of caching fails then we go to the second stage of caching, which is file caching. We store the parquet file created after performing an action by the agent. Thus, in a later period, the agent can have access to the partitioned file for executing the query. Therefore saving computational time for partitioning a table and utilizing the partitioned file which previously exists.

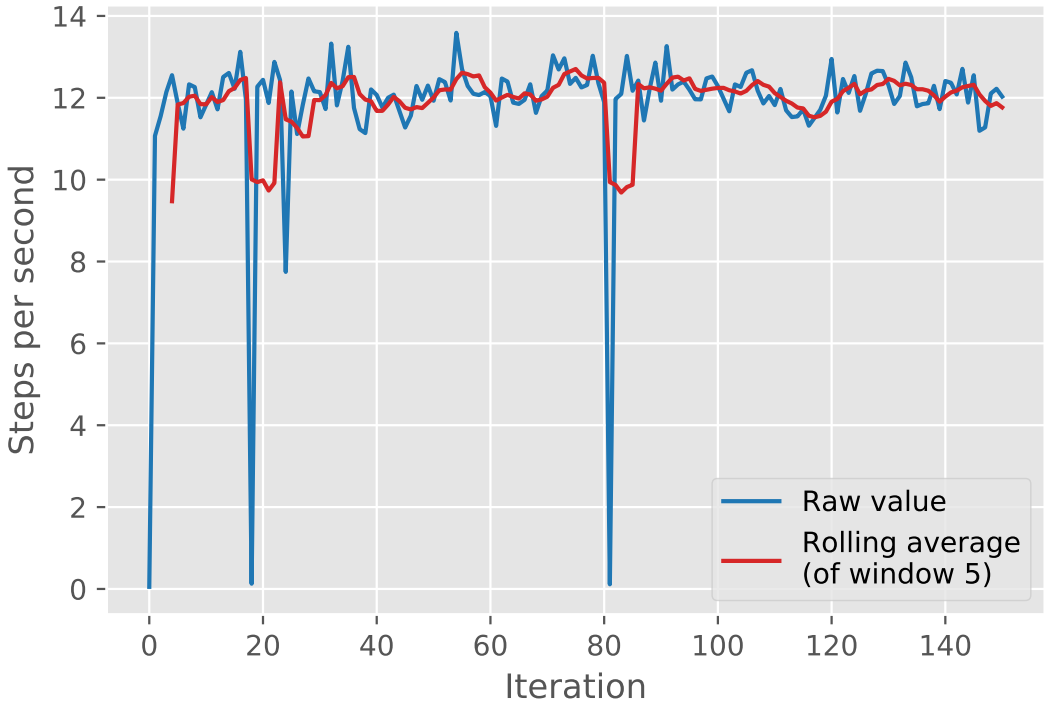
Following the above-mentioned method allows us to efficiently calculate the reward and prevents the agent from repeating a certain redundant computation and allows the agent to re-use the computation that it has performed before, leading to an improvement in computational time.

In Figure 3.4 and Figure 3.5 we see the number of steps per second taken by an agent. This includes the time for training and the time for performing the actions (partitioning and calculating the reward based on runtimes). We can see for both cases, and more strongly for the test scenarios, the agent begins without too much cached data, leading to very low steps per second, but as the training proceeds and there is more cached data, then the agent sees less cases where the steps per second are so low. These observations illustrate the benefits that caching brings for improving the training of the agents.

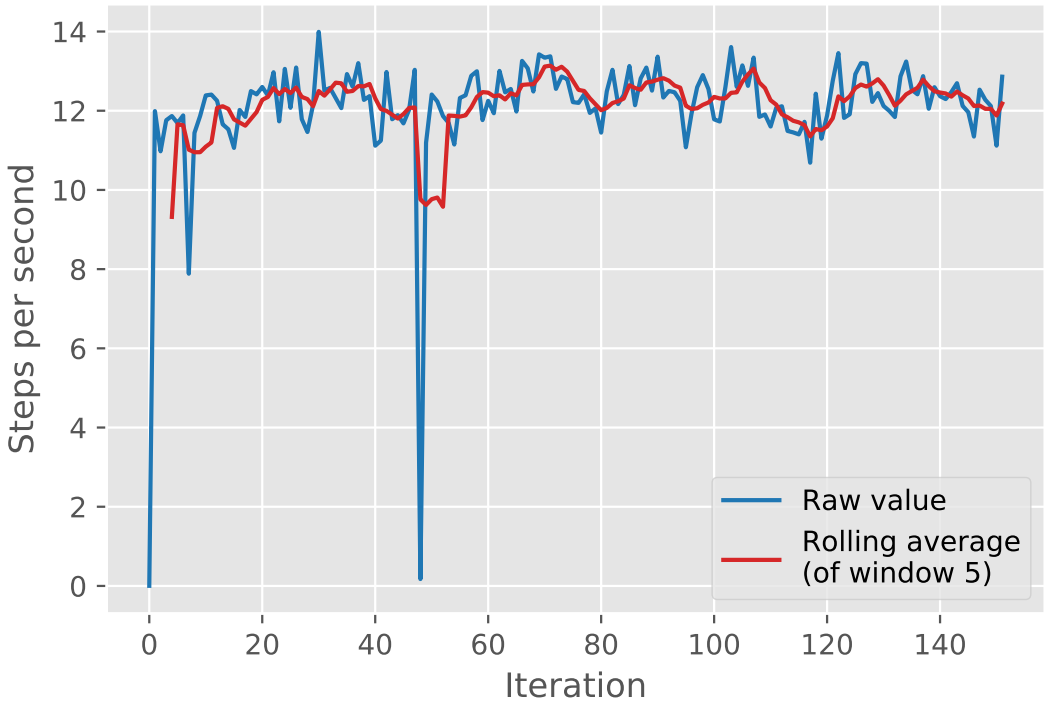
3.2.3 Model improvements

For improving the model that we are going to work on in our study we have implemented an *action mask*. In this section, we are going to present how the action mask operates for the agent and the environment.

Definition 3.3 (Action mask) *An action mask allows the agent to identify valid and invalid actions prior to taking an action on the environment.*

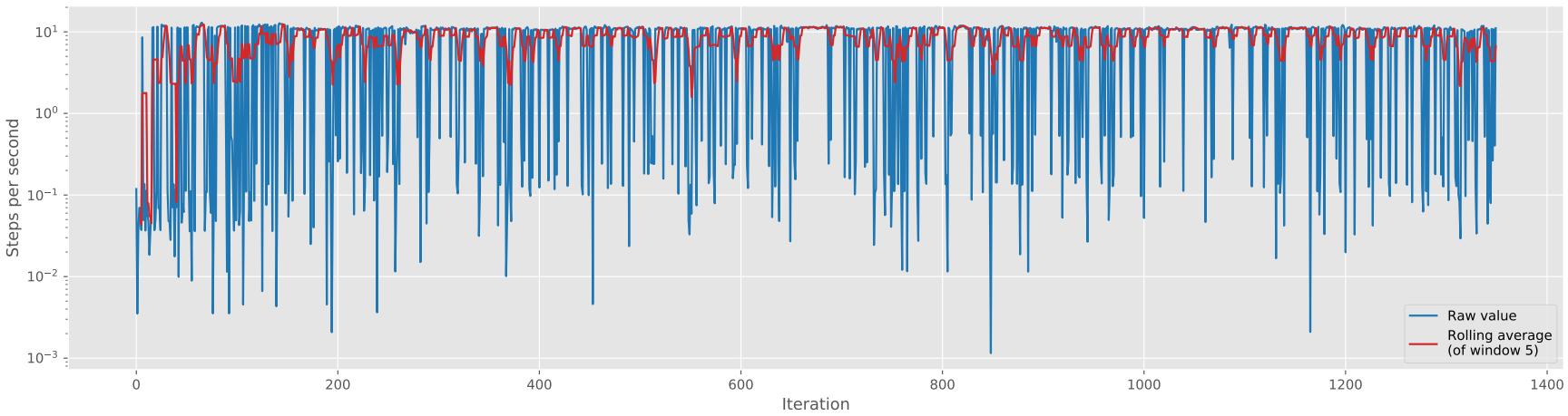


(a) Steps per second w.r.t iteration (DQN)

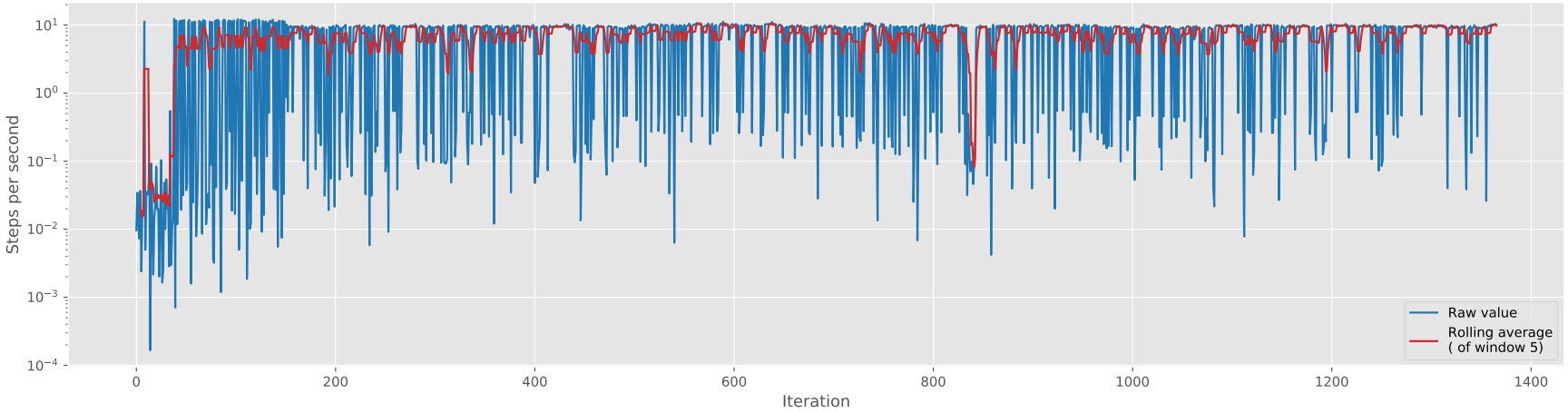


(b) Steps per second w.r.t iteration (R2D2)

Figure 3.4: Steps per second for DQN and R2D2 w.r.t iteration during testing.



(a) Steps per second w.r.t iteration (DQN)



(b) Steps per second w.r.t iteration (R2D2)

Figure 3.5: Steps per second for DQN and R2D2 w.r.t iteration during training (using log scale for y axis)

We have designed an action mask for the agent that can be applied during the training of the models. Moreover, as a second layer of verification of the actions, we have implemented an identical logic of action mask inside the environment.

The purpose of an action mask is to allow the agent to differentiate between valid and invalid actions prior to performing the action, leading to avoidance of computation that is not necessary. Since in RL, the agent learns by trial and error, and typically the training time involved in solving RL problems is very long, therefore it is crucial to save the computation and training time by avoiding redundant actions.

At first, we are going to present how the action mask looks like for an agent before starting an episode.

action mask = $\big[1\ 1\ 1\

Furthermore, for both agents we have used the action mask as a component we add to the replay buffer so we can use it in training. Specifically when we calculate the probabilities on the target network assuming greedy actions, we need to employ the action mask for the corresponding time-step. As this is not presented for the models we consider this a contribution to our work. We describe our approach in the pseudo-code snippets provided: 3.2, 3.3, 3.4, and 3.5.

Algorithm 3.2: Pseudo-code of basic DQN training

Input: Observations for timestep 0 O_{t0}, actions for timestep 0 A_{t0}, rewards R, discount D, observations for timestep 1 O_{t1}

Initialization: Retrieve a batch of input variables by sampling replay buffer

```
q_values = network(O_t0)
target_q_values= target_network(O_t1)
q_target_selector = network(O_t1)
extras = double_q_learning(q_values, A_t0, R, D, target_q_values,
  q_target_selector)
loss = losses.huber(extras.td_error, huber_loss_parameter)
return minimize(loss)
```

Algorithm 3.3: Pseudo-code of our usage of the legal action mask in DQN training

Input: Observations for timestep 0 O_{t0}, actions for timestep 0 A_{t0}, rewards R, discount D, observations for timestep 1 O_{t1}, mask for timestep 1 M_{t1}

Initialization: Retrieve a batch of input variables by sampling replay buffer

```
q_values = network(O_t0)
target_q_values= target_network(O_t1)
q_target_selector = network(O_t1)
q_target_selector = tf.where(tf.equal(M_t1, 1), q_target_selector,
  tf.fill(tf.shape(q_target_selector), -np.inf))
(Comment: We make negative infinity all invalid actions from the action
mask, so they do not get selected by mistake in the double_q_learning loss
calculation.)
extras = double_q_learning(q_values, A_t0, R, D, target_q_values,
  q_target_selector)
loss = losses.huber(extras.td_error, huber_loss_parameter)
return minimize(loss)
```

Next, we will present how the logic of the action mask is implemented inside the environment as a second layer of verification.

In our experiment, before the agent takes any action, it first checks the observation space (which is a 2D matrix). If the action is valid, then the agent executes the action, and if it is not valid, then the agent gets a negative reward of -1 and the current episode is terminated.

For instance, at the beginning of an episode, the observation space is filled with zeros in all entries of the index, and when the agent picks an action to partition a

Algorithm 3.4: Pseudo-code of basic R2D2 training

Input: Observations *O*, actions *A*, rewards *R*, discounts *D*, core state *CS*, target core state *TCS*, sequence length S, number of actions *num_actions*, n-step value *N*

Initialization: Retrieve a batch of input variables by sampling replay buffer

```
q_values, next_state = network.unroll(O, CS, S)
target_q_values, target_next_state= target_network.unroll(O, TCS, S)
greedy_actions = tf.argmax(q_values)
target_policy_probs = tf.one_hot(greedy_actions, length=num_actions)
rewards = R[:-1]
discounts = D[:-1]
loss = losses.n_step_loss(q_values, target_q_values, A, rewards, discounts,
  target_policy_probs, N)
return minimize(loss)
```

Algorithm 3.5: Pseudo-code of our usage of the legal action mask in R2D2 training

Input: Observations *O*, actions *A*, rewards *R*, discounts *D*, core state *CS*, target core state *TCS*, sequence length S, number of actions *num_actions*, n-step value *N*, mask *M*

Initialization: Retrieve a batch of input variables by sampling replay buffer

```
q_values, next_state = network.unroll(O, CS, S)
target_q_values, target_next_state= target_network.unroll(O, TCS, S)
q_values2 = tf.where(tf.equal(M, 1), q_values, tf.fill(tf.shape(q_values),
  -np.inf))
(Comment: We make negative infinity all invalid actions from the action
mask, so they do not get selected by mistake in the n_step_loss calculation.)
greedy_actions = tf.argmax(q_values2)
target_policy_probs = tf.one_hot(greedy_actions, length=num_actions)
rewards = R[:-1]
discounts = D[:-1]
loss = losses.n_step_loss(q_values, target_q_values, A, rewards, discounts,
  target_policy_probs, N)
return minimize(loss)
```

table (i.e. a column from a table), the column chosen to partition the table and all the corresponding columns of that particular table's index is replaced by 1. In this way, we have made certain that the agent is allowed to take only valid actions and can avoid taking invalid actions inside the environment also.

We believe, by applying the action mask, we can expect the agent, in the course of training to learn the behavior of avoiding invalid actions. For example, if the agent takes an action and then based on the previous experience, the agent should be able to know which action to be avoided as the next actions. This will consequently lead to improved learning for the agent by avoiding performing actions that are not valid in advance.

3.3 Summary

In this section, we have presented our research questions. Next, we presented the design of our prototype, which includes: the design of our environment, the observation space, the actions, calculation of the reward for the agent, stopping conditions for the agent in an episode, and presented the dynamics of our environment. We also presented the caching strategy that we have applied in our experiments to be more efficient. In the end, we presented how we have improved the DRL model for our study. In the following Chapter 4 on the facing page, we will discuss in detail our experimental setup.

4. Experimental setup

In this chapter, we present all the required experimental setup aspects that we have followed for our experiments and which can be adapted to replicate our experiments.

This chapter is organized in the following way:

- In Section 4.1 on the next page we present the dataset and the workload we are using for our experiment.
- In Section 4.2 on page 62 we present what file format we have applied in our experiments.
- In Section 4.3 on page 62 we present how we have calculated the reward based on query execution runtime.
- In Section 4.4 on page 62 we present the hyper-parameters of DQN and R2D2 agents.
- In Section 4.5 on page 63 we discuss how we have distinguished between train and test sets.
- In Section 4.6 on page 63 we present the hardware and software setup that has been used in our experiments.
- In Section 4.7 on page 64 we summarize the contents of this chapter.

4.1 Dataset and workload

As a benchmark dataset, we have used Transaction Processing Performance Council Benchmark™H (TPC-H)[8] for performing our experiments. The schema of the TPC-H Benchmark is shown in Figure 4.1. The benchmark consists of 8 tables and 22 queries. The (one-to-many) relationship between the tables is shown by arrows in the figure.

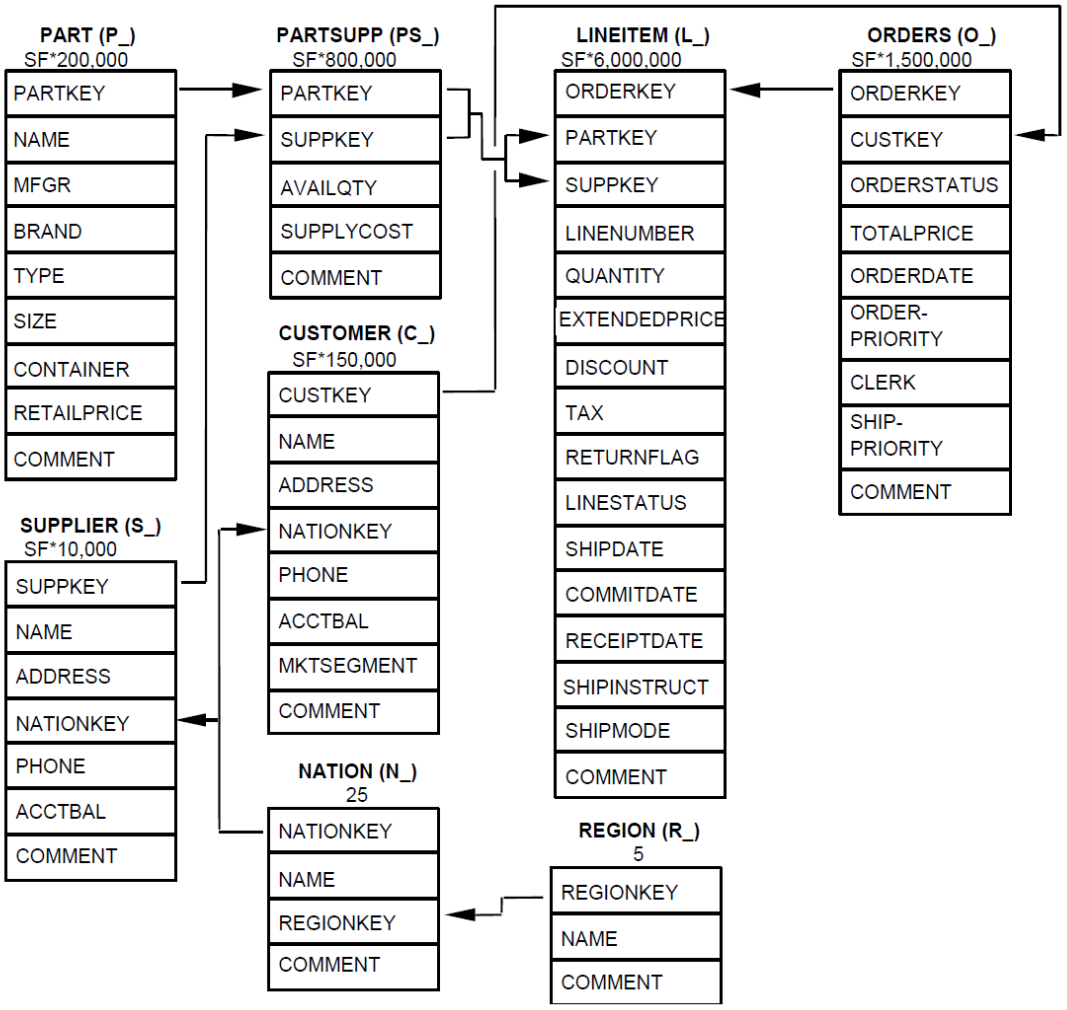


Figure 4.1: TPC-H schema[9]

In the TPC-H dataset, the supported scale factor is 1, 10, 30, 100, 300, ..., where scale factor of 1 will generate 1GB, a scale factor of 10 will generate 10 GB, scale factor of 30 will generate 300GB, scale factor of 100 will generate 1000GB and a scale factor of 300 will generate 3000GB of data. We have worked with scale factor of 0.05, which is 50 MB of data. In principle we intended to work with a scale factor of at least 1, but as this required a specialized logic to manage the exponential data growth when storing different partitions, we have reduced our observations to a much smaller data instance. We consider that our choice is comparable to the

[8] http://www.tpc.org/tpch/
[9] http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf

sample-based approach of other authors in the field [HBR20]. In future work, we should validate how the observations made for a small sample generalize to larger scale factors.

For the workload, we have used the queries that are generated by QGEN[10] which is provided by TPC. Below, we provide all the 22 queries[9] definition. We would also like to mention that we have used 21 queries for our experiments, except query number 15 which was not trivially supported in Spark.

1. Pricing Summary Report Query (Q1)
2. Minimum Cost Supplier Query (Q2)
3. Shipping Priority Query (Q3)
4. Order Priority Checking Query (Q4)
5. Local Supplier Volume Query (Q5)
6. Forecasting Revenue Change Query (Q6)
7. Volume Shipping Query (Q7)
8. National Market Share Query (Q8)
9. Product Type Profit Measure Query (Q9)
10. Returned Item Reporting Query (Q10)
11. Important Stock Identification Query (Q11)
12. Shipping Modes and Order Priority Query (Q12)
13. Customer Distribution Query (Q13)
14. Promotion Effect Query (Q14)
15. Top Supplier Query (Q15)
16. Parts/Supplier Relationship Query (Q16)
17. Small-Quantity-Order Revenue Query (Q17)
18. Large Volume Customer Query (Q18)
19. Discounted Revenue Query (Q19)
20. Potential Part Promotion Query (Q20)
21. Suppliers Who Kept Orders Waiting Query (Q21)
22. Global Sales Opportunity Query (Q22)

[10] https://github.com/databricks/tpch-dbgen

4.2 File format

We have used parquet file format in our work where data are stored in row groups, which consists of batches of columns[11].

For every column there exists a file metadata. The advantage of having metadata is that it permits data skipping based on the column values that the user is not interested in[11]. Parquet file format allows several kinds of encoding mechanisms to store the data in an efficient way[12].

Due to the effectiveness of parquet file format, the data involved in partitioning, we have used parquet file for our experiments. Parquet also includes compression mechanisms that further boost the performance gains. In future work, we could consider compression and partitioning together.

4.3 Reward calculation

As mentioned in Section 2.3.1.2 on page 23 and in Section 3.2.1.4 on page 49, the reward signal plays a central role in defining the goal for the agent by determining which action the agent would choose in each step. A slightly wrong formulation of the reward can lead to catastrophic failure in our experiments and can lead to outcomes that are not explainable. Therefore, a careful formulation of the reward is much necessary to achieve the purpose of our thesis.

As discussed in the Section 3.2.1.4 on page 49, we have chosen to adopt the reward formulation as Equation 3.1 on page 49 provided by [SSD18]. For calculating the time (in seconds) to execute the workload, we have adopted the following function from the Python library

- time.time()[13]

to monitor the runtime.

We also considered as a more domain-specific alternative the use of SparkMeasure, a specialized library for monitoring the performance of Spark [Can19], but due to issues with the library we take its adoption as future work.

4.4 Hyperparameters for DQN and R2D2

Hyperparameters play a significant role in optimizing the performance of machine learning algorithms. With the appropriate combinations of hyperparameters, the performance of the algorithms can differ considerably [FH19].

Due to the complex nature of a DRL agent, and because any RL problem requires an enormous amount of training steps, it is rather challenging (due to high computational cost) to tune the hyperparameters systematically (for example by using grid search or random search) [FH19].

[11] http://parquet.incubator.apache.org/documentation/latest/
[12] https://github.com/apache/parquet-format/blob/master/Encodings.md
[13] https://docs.python.org/3/library/time.html#time.time

In our experiments, we have used two agents from the Acme framework [HSA+20], DQN [MKS+13, MKS+15] (in Acme, DQN is implemented with some improvements that have been made following the first publication of the original DQN agent) and R2D2 [KOQ+18]. Both of these agents, like any other agents in DRL, have hyperparameters that can be tuned.

In our experiments for DQN and R2D2, we have mostly kept all default hyperparameter settings offered by ACME. For DQN we have set to the epsilon to 0.05. The network we used for DQN is also what is provided by ACME, which is composed of 3 convolutional neural networks followed by a rectifier linear unit activation layer after each convolutional layer. The convolutional layers consist of 32, 64, and 64 filters respectively, of shape 8×8, 4×4, and 3×3 respectively for each convolutional layer. Following this network architecture, Acme adopted the dueling network architecture [WSH+16] as the second component of the DQN architecture. For R2D2, the hyperparameter setting that we have adopted is as follows: burn_in_length as 0, trace_length as 5, and replay_period as 1. The network of R2D2 is similar to DQN except with an LSTM layer (consisting of 512 units) before the dueling network, the other hyperparameters we have kept as provided by Acme.

Below, we provide a table for hyperparameters that are both common in DQN and R2D2 in Table 4.1 on page 65, hyperparameter specific to DQN in Table 4.2 on page 65 and hyperparameters specific to R2D2 in Table 4.3 on page 66.

4.5 Train and test set

For any machine learning algorithm, it is essential to have a training set of instances, which will be utilized for learning, and test set instances that will be applied for testing the performance of the algorithm. For this purpose in our experiment we have trained the model for 10 consecutive iterations and tested the model for the next iteration and continued to train for the next 10 iterations, i.e., every iteration numbers 1, 11, 22, 33, 44 ... are test instances and the rest are training iterations. Overall, we train for 1500 iterations/episodes.

4.6 Experimental Environment

We have performed our experiment on a system with the following characteristics:

- Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-112-generic x86_64)
- 72 Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz cores
- 376 GBs of RAM
- Storage: HDD AVAGO MR9361-8i, with 24 TB

The software configurations concerning the DRL agent are presented below:

- dm-acme 0.2.0

- dm-env 1.4
- dm-reverb 0.2.0
- dm-sonnet 2.0.0
- dm-tree 0.1.5
- tensorflow 2.4.1
- trfl 1.1.0

From these libraries we can specifically note that dm-acme provided the basic agent implementation, dm-env the environment wrappers, dm-reverb the server for replay buffers, dm-tree provided tree structures that were used for the R2D2 agent, trfl was employed for some utilities such as epsilon greedy exploration, and finally dm-sonnet and tensorflow were used to implement the neural network architectures and functions for their training.

For data management, we have used pyspark 3.1.1 in single-process mode.

4.7 Summary

In this chapter, we have presented the dataset and the workload that we are using in our work, we gave a brief overview of the file format that we are using. Then we have discussed how we have calculated the reward based on the execution runtime for the workload. Furthermore, we have presented the hyperparameters that are tunable for the agents (DQN and R2D2). We have briefly explained how we have divided the test set from the training set and the importance of it. We concluded this chapter by providing the hardware and software configuration that has been used for our experiments.

Table 4.1: Common hyperparameters in DQN and R2D2 [HSA+20].

Hyperparameter	**Description**
Batch size	Size of the batches from the replay buffer used by the learner.
Prefetch size	Number of experiences to transfer from the replay buffer
Target update period	The frequency of updating the parameters of the target network with the parameters of the policy network.
Sample per insert	Number of experience tuples to be extracted from the replay buffer every time an experience tuple is added to the replay buffer.
Min replay size	Minimum number of experience that needs to be stored in the replay buffer before training starts.
Maximum replay size	Maximum number of experience that can be stored in the replay buffer.
Importance sampling exponent (β)	For correcting the bias that prioritized experience replay brings [SQAS16].
Priority exponent (ω)	Exponent value for prioritized experience replay [SQAS16].
Epsilon (ϵ)	The value of ϵ in ϵ-greedy w.r.t exploration and exploitation.
Learning rate	Learning rate of the optimizer used.
Discount (γ)	The value of the discount factor in the return.
Optimizer	Optimizer used to update the weight of the neural network w.r.t the loss

Table 4.2: Hyperparameter for DQN [HSA+20].

Hyperparameter	**Description**
N-step	It defines how many steps taken by the agent will be considered as one transition.

Table 4.3: Hyperparameters for R2D2 [HSA+20].

Hyperparameter	**Description**
Burn in length	Length of the replay sequence that will be unrolled but not considered for the training.
Trace lengths	The rest part of replay sequence which is used for updating the network's parameter.
Replay period	When adding experiences in the replay buffer, the time step followed to add the agent's experience i.e. the sequence.
Learning rate	Learning rate of the optimizer used.
Max priority weight	Maximum priorities for the experience tuple in the replay buffer.

5. Evaluation and Results

In this chapter, we are present the experiments and their results, along with a discussion that we have performed w.r.t to our research questions.

This chapter is organized in the following way:

1. In Section 5.1 we present two experiments we have carried out by using R2D2 and DQN agent for overfitting to the training scenarios
2. In Section 5.2 on page 78 we present experiments for evaluating how a DQN agent generalizes beyond the training experience.
3. In Section 5.3 on page 85 we present experiments to understand how an R2D2 agent generalizes beyond the training experience.

5.1 Over-fitting to training examples

Experiment 1

In this experiment we will answer to our first research question w.r.t DQN [MKS+13, MKS+15]:

1. To what extent is DQN, a discrete DRL model, able to achieve good rewards on a partitioning use case when overfitting to the training scenarios?

Experiment set-up

We performed our experiment using TPC-H data with a scale factor 0.05 on the workload containing 21 queries.

For this experiment, we have divided the dataset into train and test sets in the following way: 10 consecutive iterations are for training the model, which we then

interleave with one testing iteration (where the model acts greedily, as opposed to training where the model also explores).

We have recorded both the episode return (i.e. cumulative reward for the given episode), the final runtime (in seconds) for executing the workload on the given partition obtained at the end of the episode, and the final runtime gain ratio (i.e. how much faster was the agent to find a good partitioning and to execute the workload on the partitioned data compared to the workload execution time for non-partitioned data) in the given episode.

Our hypothesis is that the agent will learn a good reward, but we are uncertain if convergence will be possible.

For our parameters we used a batch size of 64, a target update of 500, a minimum replay size of 1000, an importance sampling exponent of 1.0, a priority exponent of 0.6, epsilon of 0.05, learning rate of 0.00001, Adam as an optimizer, gradient clipping of 10, a discount of 0.99 and an n-step of 5. These hyper-parameter settings are small deviations from the defaults of Acme, which we found represented a competitive configuration for our application.

Following, we present the results along with our discussion.

Results

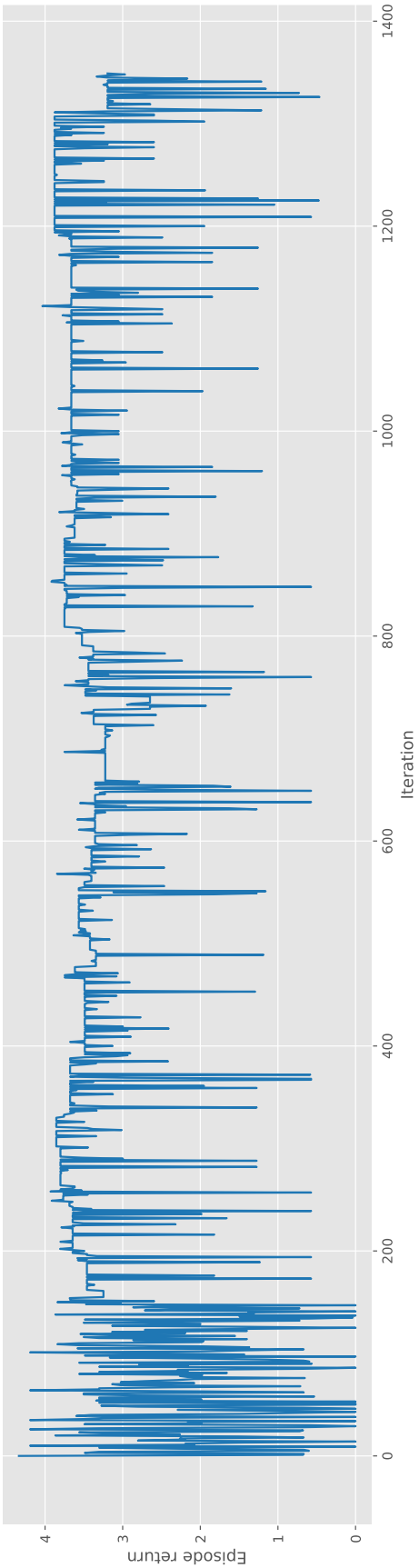
In Figure 5.1 on the next page and in Figure 5.2 on page 70 , we report the results regarding the DQN agent w.r.t cumulative episodic return and final workload execution runtime for training and testing per iteration/episodes, respectively

In Table 5.1 we provide the statistics related to training and testing for DQN agent.

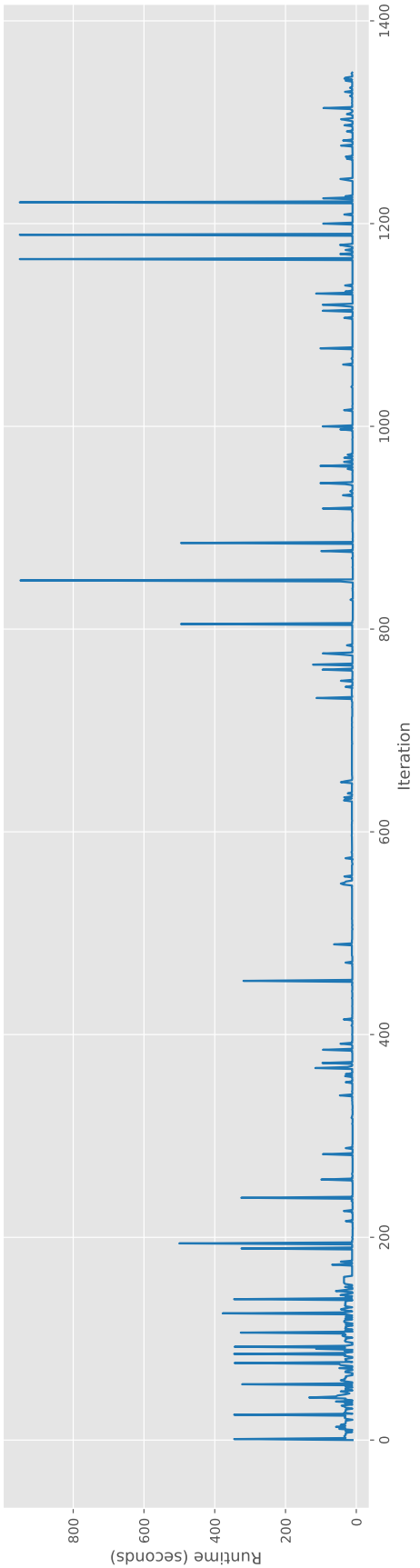
Table 5.1: Episode return and final runtime for DQN.

	Train	**Test**
Iteration	1350	151
Lowest return (cumulative)	0.0	1.139532
Highest return (cumulative)	4.337047	3.877811
Lowest final runtime (seconds)	9.968987	9.968987
Highest final runtime (seconds)	951.23033	52.441141
Final runtime gain (ratio)	1.742456	1.742456

Discussion

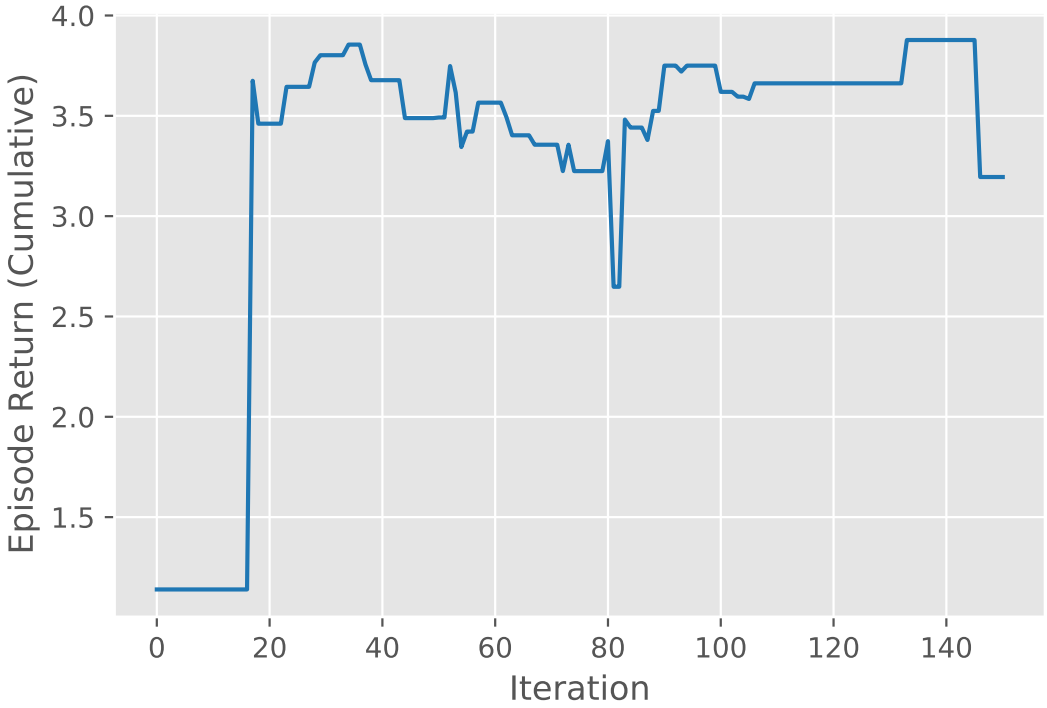


(a) Cumulative episodic return at the end of the episode per iteration.

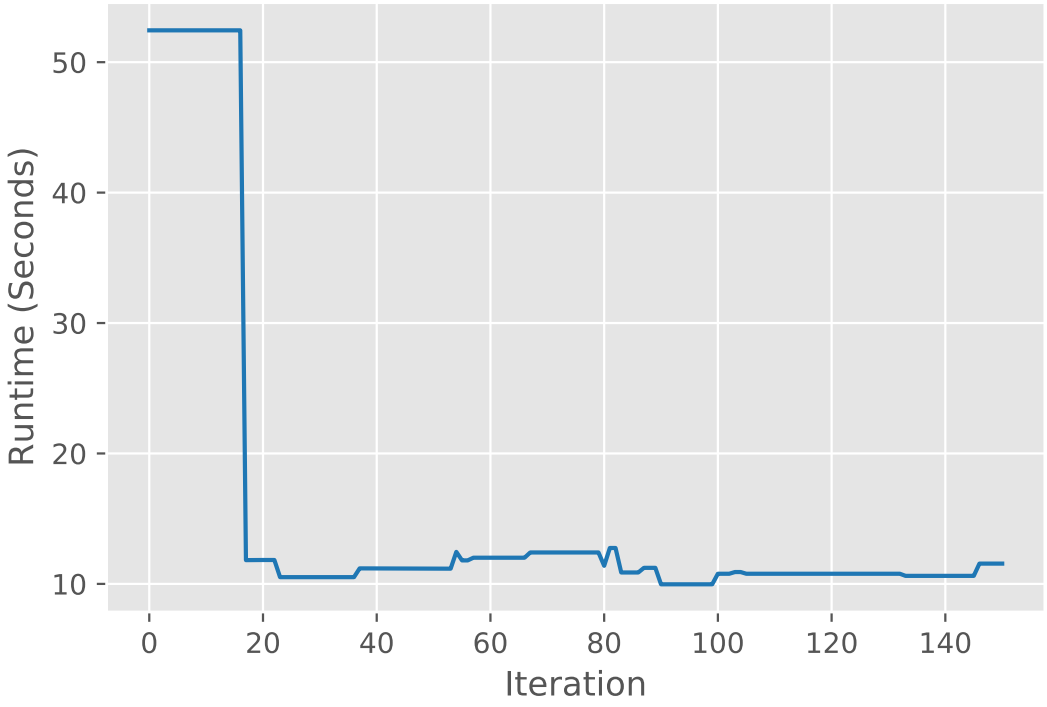


(b) Final runtime for workload execution per iteration.

Figure 5.1: Cumulative episodic return and runtime for workload execution per iteration for DQN (training dataset).



(a) Cumulative episodic return at the end of the episode per iteration.



(b) Final runtime per iteration.

Figure 5.2: Cumulative episodic return and final runtime for workload execution per iteration for DQN (test dataset).

In our experiment for *training* (Figure 5.1a on page 69) w.r.t episode returns, we observed a pattern of episodic reward oscillating in a chaotic way in the beginning of the training period. This is because, before the 167th iteration, the agent has not started training and instead the replay buffer is simply being filled with the minimum experience necessary for training. After around the 167th iteration we can observe less oscillation, since after the 167th iteration training of the model starts. The highest reward (i.e., the optima) achieved during training is 4.337047 by the agent. As the number of iterations increases, we were supposed to see more stable training but the agent does not converge to the optima it has seen before. We hypothesize that this happens since the agent keeps on exploring the environment even after finding the optimal actions, and this leads to finding sub-optimal solutions, similarly the target network update has some effect in destabilizing previous knowledge. We reason that higher number of iterations would lead to a more stable training, where the agent should converge to the optima. But this needs further studies.

During *testing* (Figure 5.2a on the preceding page) we see a similar kind of pattern as in the training scenarios, but relatively more stable. During the first few iterations of testing, the episode return is 0, since the agent has not learned anything yet, but as soon as training starts, the test iterations shows improvements with time to obtain a better solution. However, in this case the agent is unable to reach the optima seen during training, where the highest reward achieved during the test is 3.877811. This specifically shows that the optima reached was not learned at any point.

In this experiment, we also kept record of the final *workload execution runtime* i.e. the runtime for executing the workload on the final partition of the dataset in the given episode. In Figure 5.1b on page 69 during training we can observe fluctuations in the runtime for around the first 167th iteration, however after the 167th iteration when the model starts to train, we can see the agent shows less oscillating behaviour and reaches the lowest runtime of 9.968987 seconds, but is unable to stay there. We observe the same behaviour during testing where the agent also reaches the lowest runtime 9.968987 but shows less oscillation and the agent converges. We hypothesize that during training the agent is unable to stay at the optimal runtime since the agent continues to explore the environment. In spite of these observations, it is notable that the agent decreases the runtime while training from the works case scenario.

We recognize that the way we have adopted the reward definition that we have provided in this Section 3.2.1.4 on page 49 is unclear and misleading solely because we are reporting the cumulative reward at the end of the episode (also it is worthy to notice that a reward of 3.9 would mean, that the agent has found a partition that is 4.9 times better for executing the workload compared to the baseline i.e the non-partitioned data). Unfortunately, we have reported the cumulative reward at the end of the episode and not at every step (i.e., the action taken by the agent). For instance, if the agent receives a reward of 3.9 it can be very much possible that the agent did not receive any rewards in the other steps of the episode except the last one, or it can also be possible that the agent only earned a reward at the first step and then received a reward of zeros in all other steps.

For this purpose, to alleviate this limitation, we have resolved to report the final runtime gain by the agent. The runtime that we have reported in our experiment informs us that at the end of an episode how much faster we are compared to the non-partitioned baseline w.r.t the workload execution runtime.

In Figure 5.4 on the facing page we can observe that both in the train and test dataset, the agent is able to produce a partitioning of size 6 that is around 1.74 times faster for executing the workload on the partitioned data compared to the non-partitioned data.

In Figure 5.3 we can observe that the agent has approximately converged in its learning, and the loss value (presented in Section 2.3.3.1 on page 33) for the DQN stabilizes with slight oscillations in the end, indicating there is still room for improvement.

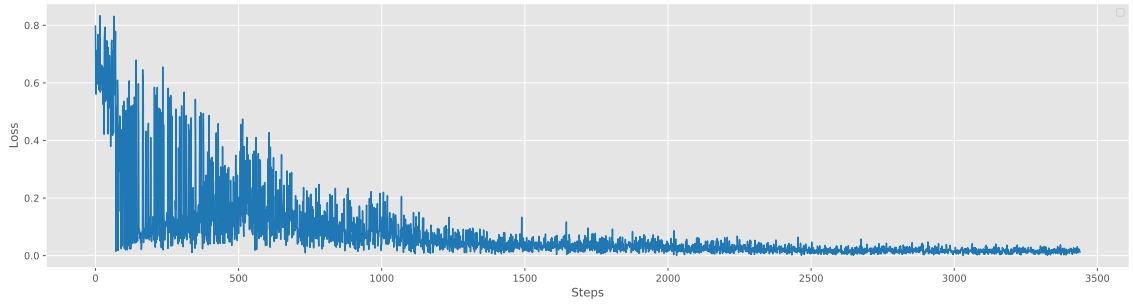


Figure 5.3: Loss for DQN agent.

In this experiment we can conclude that the agent is able to reach a competitive solution that is 1.74 times faster than the default during training and testing but since the agent has not converged completely, we hypothesize that with further training the agent and with hyperparameter tuning, the agent will converge to the best partitions it has seen so far.

Experiment 2

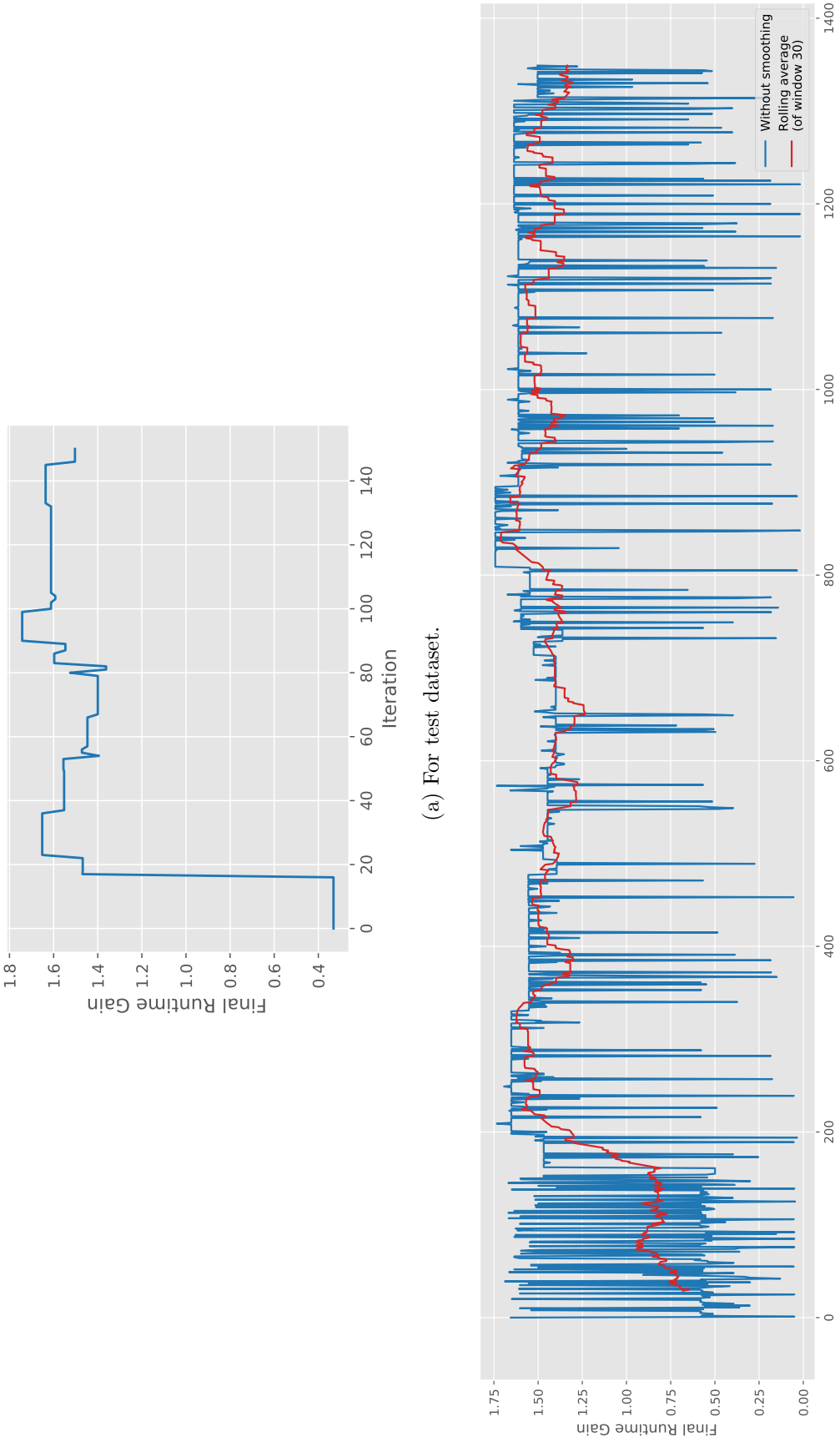
In this experiment we will answer to our first research question w.r.t R2D2 [KOQ+18]:

1. To what extent is R2D2, a discrete DRL model, able to achieve good rewards on a partitioning use case when overfitting to the training scenarios?

Experiment set-up

The experimental setup we followed to answer this research question is identical to our Experiment 1 set up. We used a burn-in length of 0, a replay period of 1, and a trace length of 5. These are hyper-parameter values that depart from those defined by default in Acme, but that we found to be competitive for our application. We also used a discount of 0.997.

Our first hypothesis for this experiment is that R2D2 will be more competitive than DQN, given that R2D2 is a specialization of DQN. However, we also consider that given the limited number of steps in the sequence, it is possible that R2D2 performs similarly to DQN.



(a) For test dataset.

(b) For train dataset.

Figure 5.4: Final runtime gain ratio for DQN w.r.t baseline.

Results

In Figure 5.5 on the next page and in Figure 5.6 on page 76 , we report the results regarding the R2D2 agent w.r.t cumulative episodic return and final workload execution runtime for training and testing per iteration, respectively

In Table 5.2 we report the statistics related to training and testing for R2D2 agent.

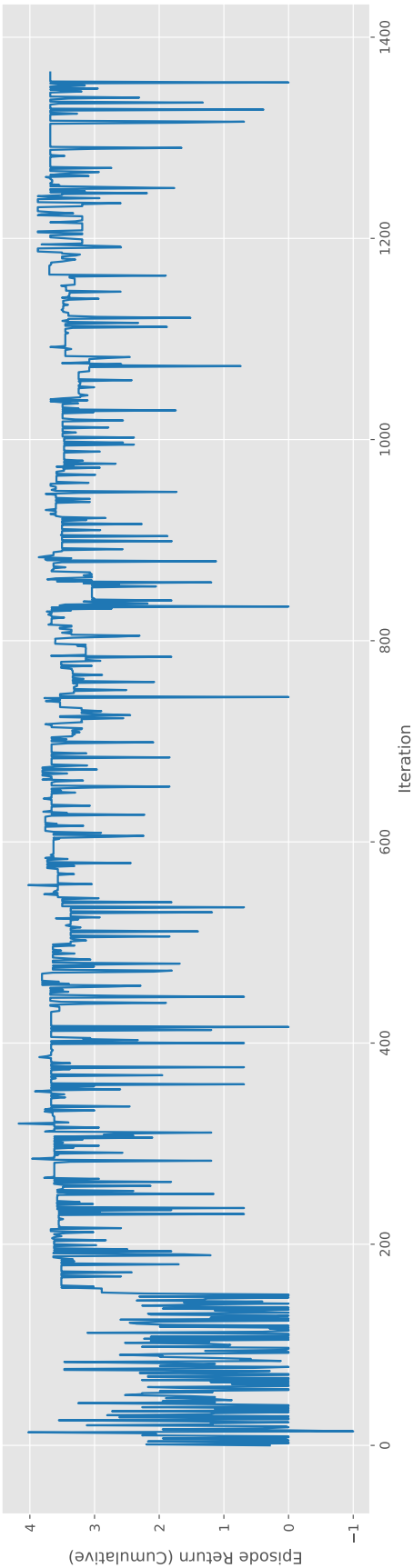
Table 5.2: Episode return and runtime for R2D2.

	Train	**Test**
Iteration	1366	152
Lowest return (cumulative)	-1.0	0.0
Highest return (cumulative)	4.174329	3.877621
Lowest final runtime (seconds)	9.954836	10.009725
Highest final runtime (seconds)	1054.024521	152.91884
Final runtime gain (ratio)	1.744933	1.735364

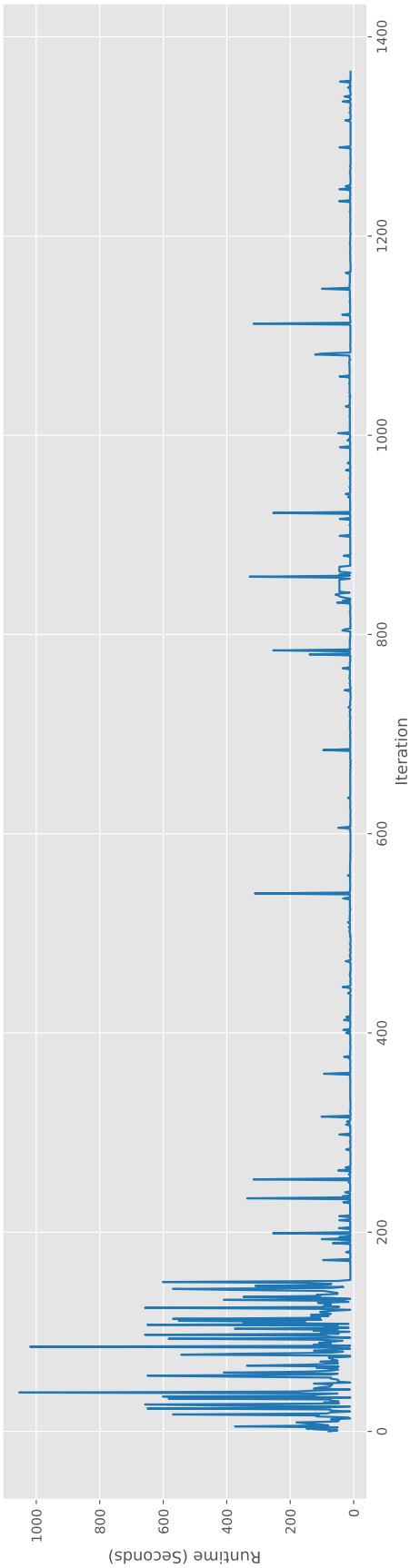
Discussion

In our experiment for *training* w.r.t episodic return (5.5a on the facing page) we observe high oscillation at the beginning of the iteration till around 167th iteration. As discussed previously, this happens since the training of the agent starts after the replay buffer is filled with the experiences gathered by the agent. After around the 167th iteration, the model starts to train. We can observe after the 167th iteration the oscillation to be less prominent compared to the start of the experiments. The highest return the agent could achieve is 4.174329 which occurs at the 357th iteration. As training continues we were supposed to see less fluctuation in the return value but the agent fails to stay at the optima it has already seen. We hypothesize that this happens due to exploration of the environment by the agent. The agent continues to explore the environment and eventually finds more sub optimal solutions. Similar to DQN, the R2D2 agent can also show some instability in its training due to the target network updates.

Also, in Table 5.2 we can observe that during training the agent received a negative reward of a -1.0, which we have not observed in any of our experiments so far. The explanation of this is that we have provided -1 reward if time-out happens, or the agent takes an invalid action, both of them will lead to resting the environment to initialization state. Therefore, we hypothesize that during an episode, the agent either performed an invalid action or time-out happened, and the agent

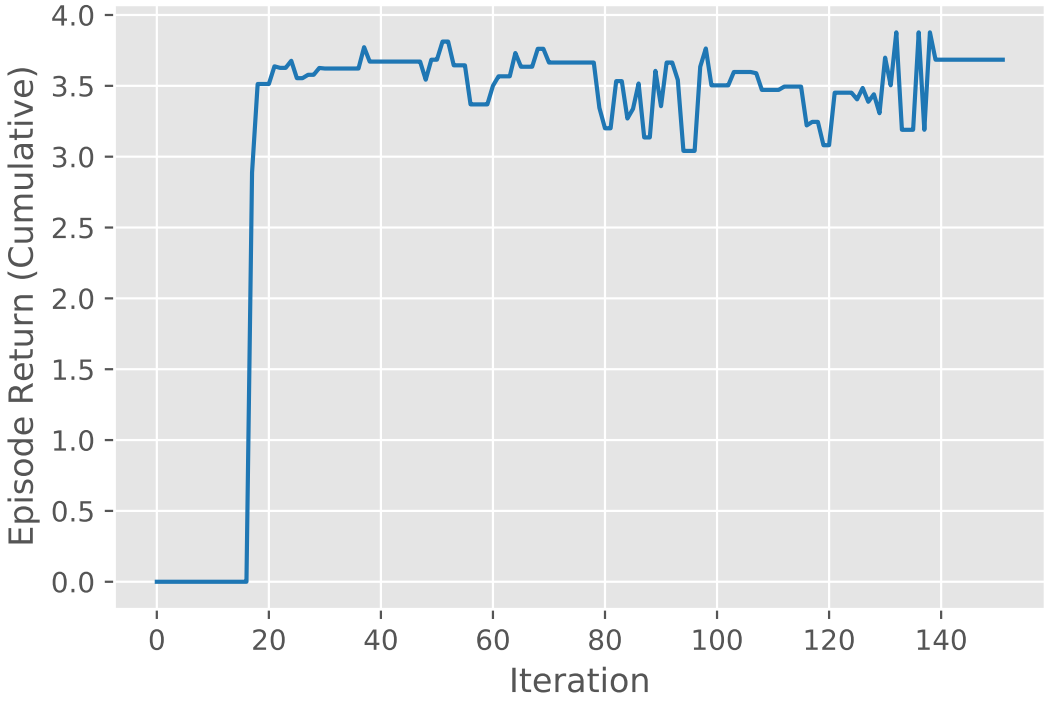


(a) Cumulative episodic return at the end of the episode per iteration.

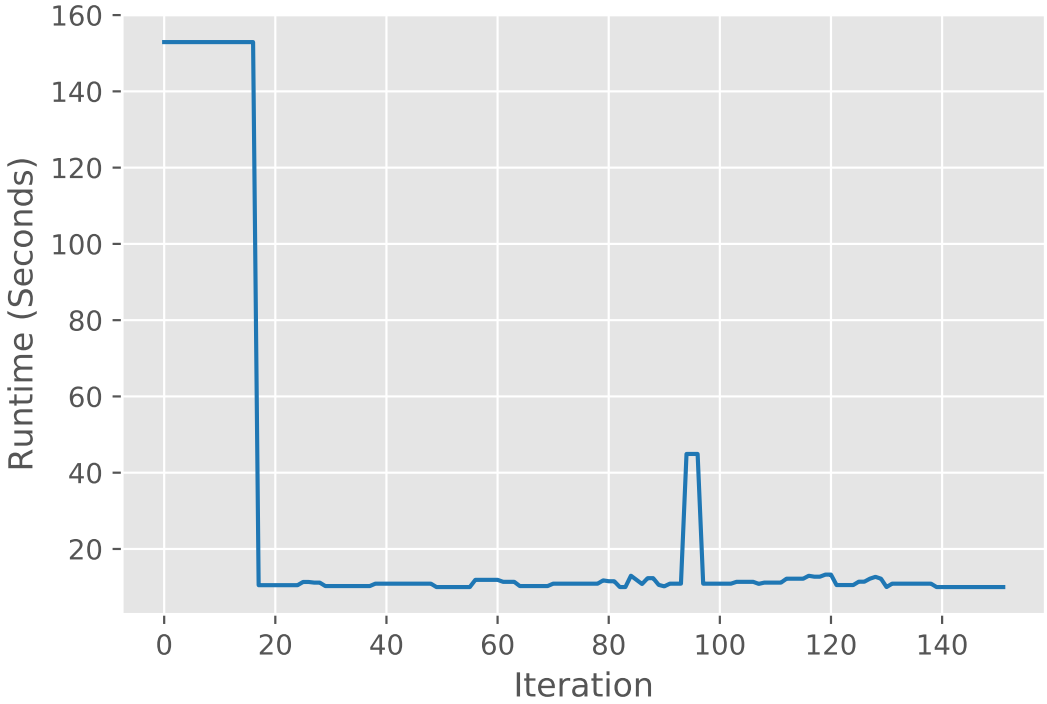


(b) Final runtime for workload execution per iteration.

Figure 5.5: Cumulative episodic return and runtime for workload execution against iteration for R2D2 (training dataset).



(a) Cumulative episodic return at the end of the episode per iterations.



(b) Final runtime per iterations.

Figure 5.6: Cumulative episodic return and final runtime for workload execution per iteration. for R2D2 (test dataset).

received a negative reward and the episode was terminated (explained previously in Section 3.2.1.6 on page 50).

During testing phase (5.6a on the preceding page) we observe similar kinds of pattern, for the first few test iteration the reward is 0, but as soon as training starts the cumulative reward per iteration in an episode starts to improve. The highest return received by the agent is 3.877621, seen first at iteration number 1321. In test scenario the agent is unable to achieve the highest reward seen during training. We hypothesize that this is due to the limited number of iterations we had for training.

Apart from the cumulative return per episode, we also recorded final *workload execution runtime* i.e. the runtime for executing the workload in the final partition in the given episode. During training (Figure 5.5b on page 75) the lowest runtime was 9.954836 seconds, seen first at iteration number 982. At the beginning of the training period, the runtime oscillates until around the 167th iteration, which reflects that the experience gathered by the agent was only to fill the replay buffer and that the training has not started yet. After the training starts, we can observe less oscillation w.r.t runtime and the agent seems to become more stable with some spikes and fluctuations in the end. During the test scenarios, the runtime is 10.009725, which is almost similar to the lowest runtime seen by the agent during the training.

As mentioned and discussed before in Experiment 1, comparing the performance of the agent w.r.t to the baseline with reward value is misleading. Therefore for this experiment we also reported the final runtime gain by the agent, both in train and test scenarios.

In Figure 5.8 on page 79 we can observe that during the training the agent was able to find solution that is 1.744933 times faster than the default partitioning, and in testing phase the agent was able to find solution that is 1.735364 times faster than the default.

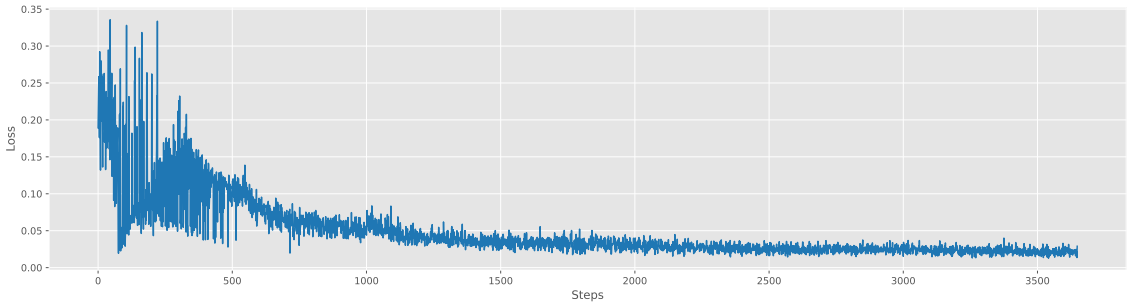


Figure 5.7: Loss for R2D2 agent.

In Figure 5.7 we can observe that the loss value for the agent decreases and continues to decrease. The agent seems to have approximately converged but with slight fluctuations still being there. We hypothesize that the agent does not converge completely and stabilizes since we have stopped the training around 1500 iterations, which can be mitigated by performing the experiment for a longer period of time.

In this experiment we can conclude that the agent was able to find good partitioning for executing the workload which is around 1.7 times faster (both for test and train

scenarios) than the default partitioning (i.e., no-partitioned data). However, as we can observe from the loss plot of the agent, the learning process has not converged completely and still continues to oscillate. We believe a higher number of training periods would allow the agent to become stable and converge, and the model can be improved more.

The winning actions that achieved the highest cumulative reward in an episode for R2D2 in this experiment are: l_linenumber, o_orderpriority, p_brand, r_name, c_mktsegment and n_nationkey.

In Table 5.3 we provide a summary comparison of the statistics for DQN and R2D2 when overfitting to the training scenario.

We can conclude from both Experiment 1 and Experiment 2, that DQN and R2D2 are able to find similar solutions that provide around 1.7 times faster solution for executing the workload and to find a partitioning that brings improvements, as compared to the baseline i.e., non-partitioned data. Besides, given the extremely large optimization space (i.e all possible partitions) which is 4654272, it is as good result that agents can reach a better solution without any external supervision (except the reward signal) and for a limited number of training iterations.

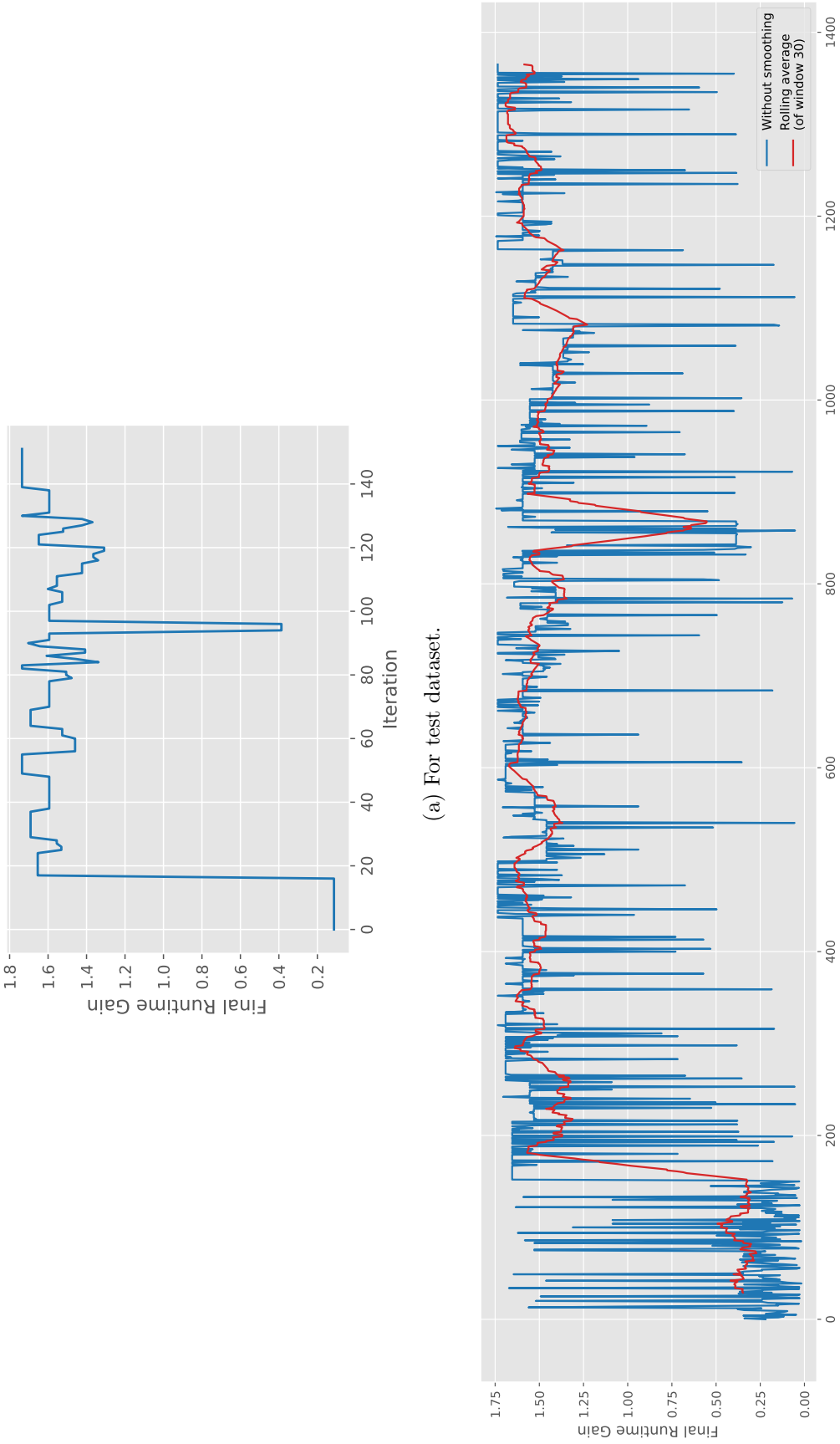
Table 5.3: Summary comparison of **RQ1** for DQN and R2D2.

	DQN		**R2D2**	
	Train	**Test**	**Train**	**Test**
Iteration	1350	151	1366	152
Lowest return (cumulative)	0.0	1.139532	-1.0	0.0
Highest return (cumulative)	4.337047	3.87781	4.174329	3.877621
Lowest final runtime (seconds)	9.968987	9.968987	9.954836	10.009725
Highest final runtime (seconds)	951.23033	52.441141	1054.024521	152.91884
Final runtime gain (ratio)	1.742456	1.742456	1.744933	1.735364

5.2 Generalizing basic discrete DRL agent (DQN) beyond training example

Experiment 3

In this experiment we will answer to our second research question:



(a) For test dataset.

(b) For train dataset.

Figure 5.8: Final runtime gain ratio for R2D2 w.r.t baseline.

1. To what extent is a traditional discrete DRL algorithms (such as DQN) able to generalize beyond training experience on the partitioning use case?

Experiment set-up

For the generalization of DQN [MKS+13, MKS+15], we have performed this experiment on TPC-H data with a scale factor of 0.05 and the corresponding workloads. For this experiment, we have separated the test, validation and train set as follows: Every 11, 21, 31, 33, 44, ... number episode are for the test set. Every 12, 22, 32, 42, 52, ... number episodes are for validation set and the rest of the episodes are training set.

We have trained our model with queries: Q1 to Q14, for validation we used the same queries: Q1 to Q14 (overfitting, but showing the greedy actions) and for test we have considered Q16 to Q22 (the query descriptions are presented in Section 4.1 on page 60). For testing we also use the greedy actions of the agent.

In terms of hyper-parameters, we used the same ones for the previous DQN experiment.

Results

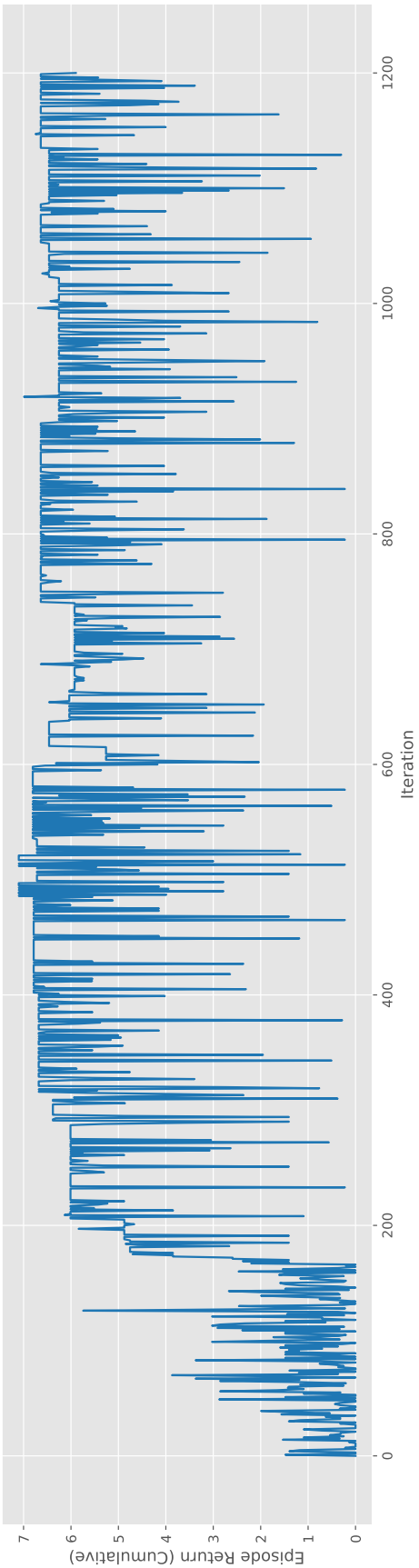
In Figure 5.9 on the next page, Figure 5.11 on page 83, and in Figure 5.12 on page 84 we report the result of training, validating and testing the agent DQN respectively. We also report the runtime gain ratio w.r.t the baseline in Figure 5.10 on page 82, since as we have discussed in Experiment 1, the episodic return is not an easy-to-interpret metric to measure the performance of the agent.

In Table 5.4 on page 85 we report the statistics related generalization of DQN agent.

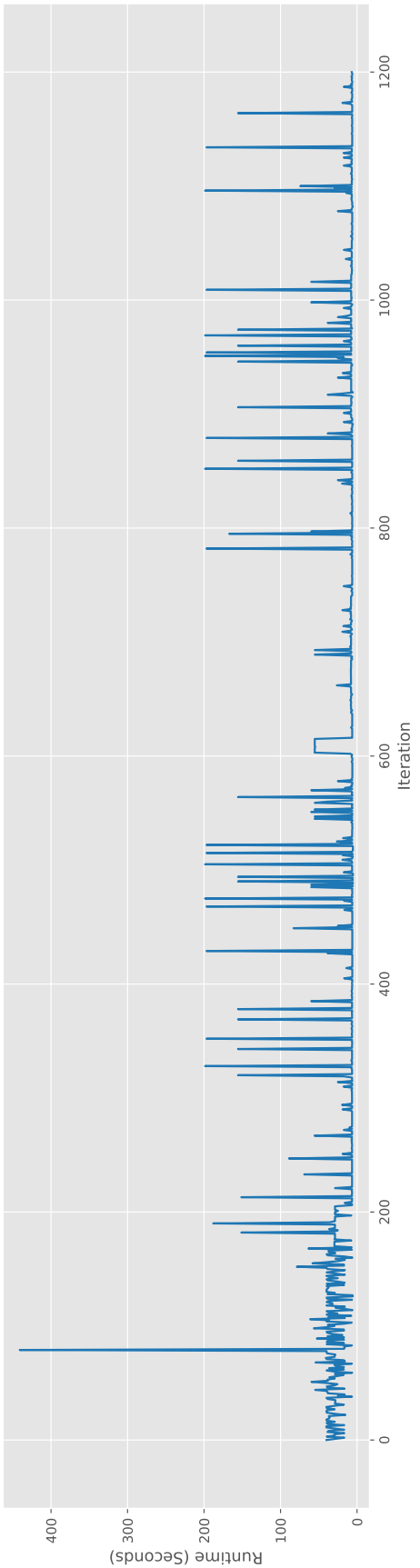
We hypothesize that the agent will not be able to generalize well since there is not enough information about the queries provided in the feature space. However, we also reason that since the TPC-H queries within themselves have a certain similarity in their access patterns, it is possible that the agent finds good partitions for the held-out queries.

Discussion

During training, we can observe that the final runtime gain ratio increases to the highest 0.786598, which indicates the agent is not able to perform better than the baseline, however, as we can observe in the validation plots, the agent performs better but it is not stable and has extensive oscillations. (For our case, validation is overfitting to the training scenarios as we have performed in Experiment 1 and Experiment 2). For testing, we used Q16 to Q22, and to our surprise, the agent's performance increases, and the final runtime gain ratio value is 4.276841 but the agent is incapable to stay there and is very much unstable. We firmly believe this occurs due to the following reason: one limitation that we have in our environment is that we do not represent such information concerning the query in the observation space. Therefore, the agent does not hold information regarding the query. Considering the agent could not observe data regarding the query but still was able to find a partition that is 4.27 times better than the baseline, we hypothesize that this

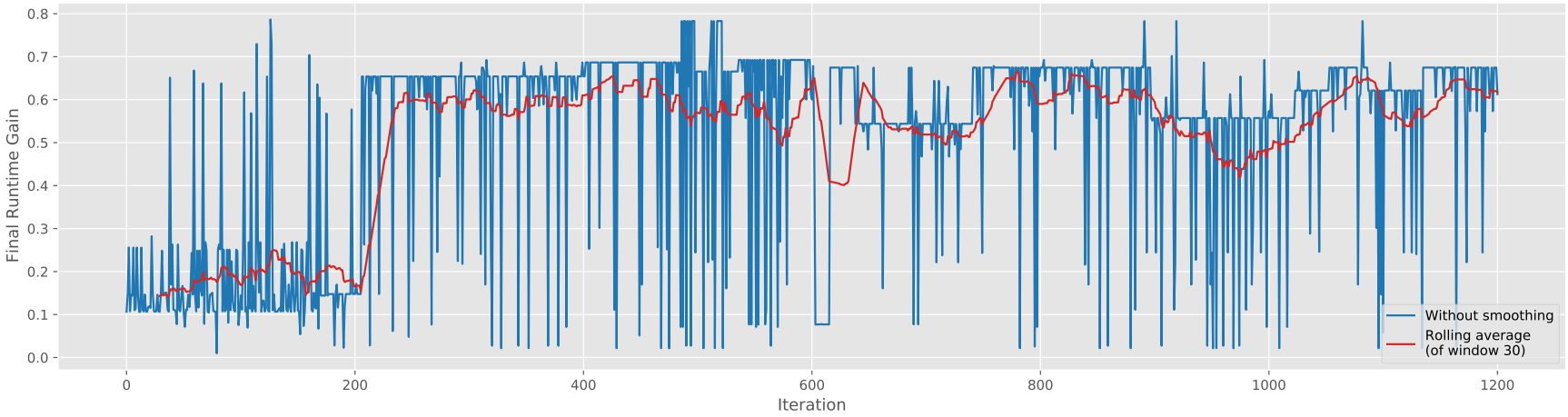


(a) Episodic return per iteration.



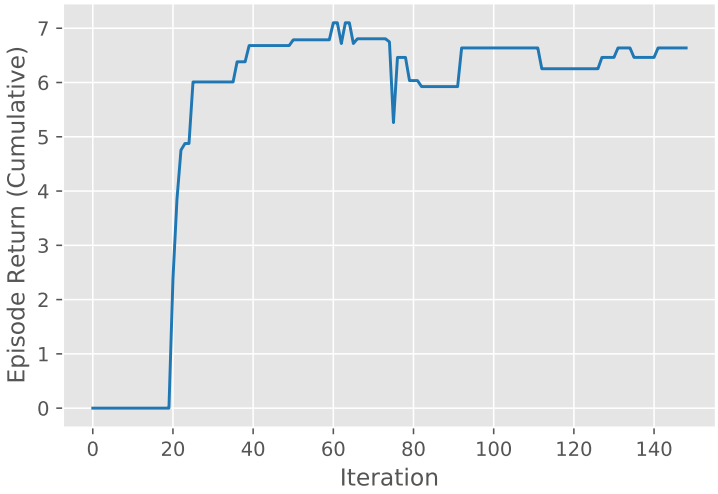
(b) Final runtime per iteration.

Figure 5.9: DQN generalization results for train data w.r.t episodic return and final runtime for workload execution per iteration.

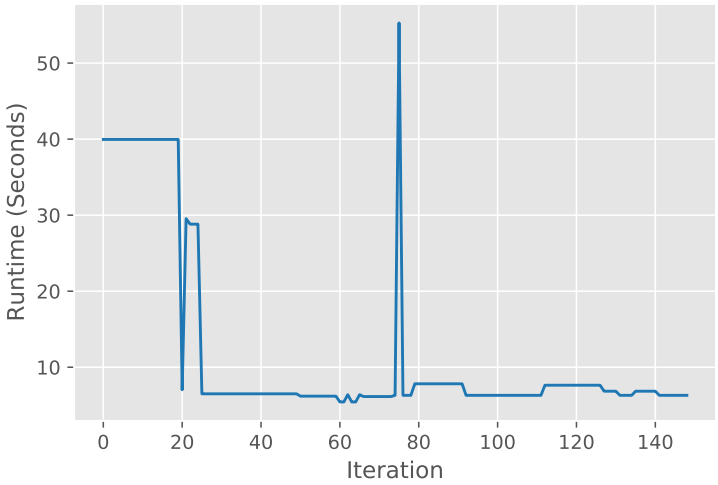


(a) Final runtime gain ratio w.r.t baseline.

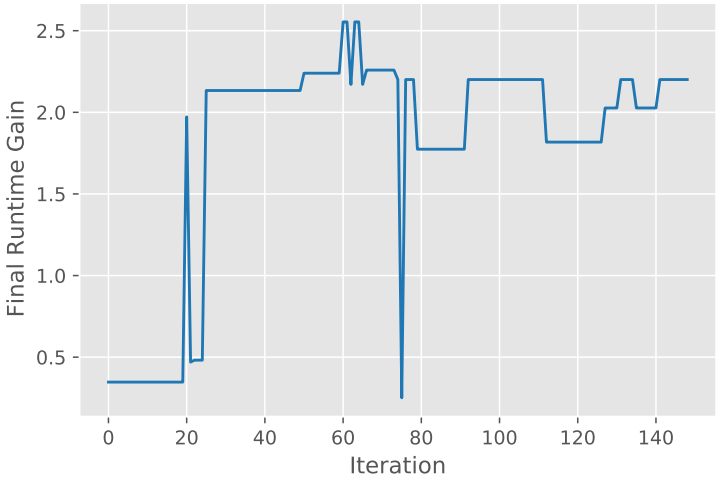
Figure 5.10: DQN generalization results for train data w.r.t final runtime gain.



(a) Episodic return per iteration.

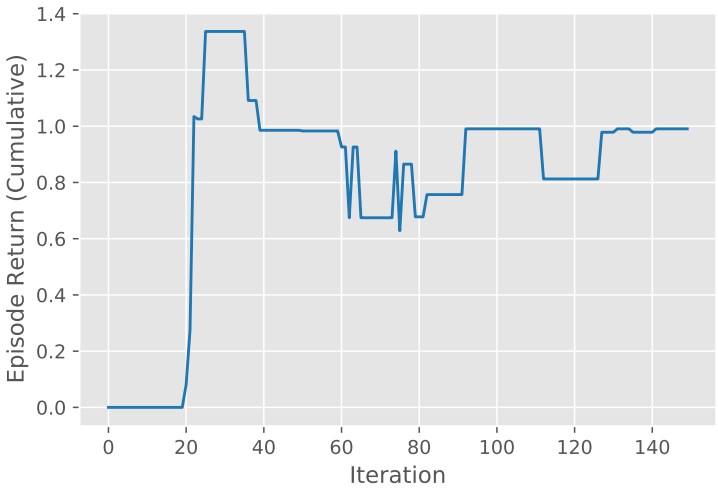


(b) Final runtime for workload execution per iteration.

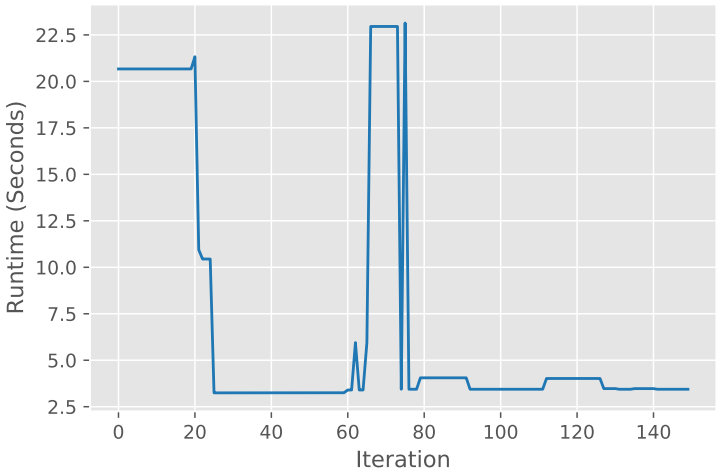


(c) Final runtime gain ratio w.r.t baseline.

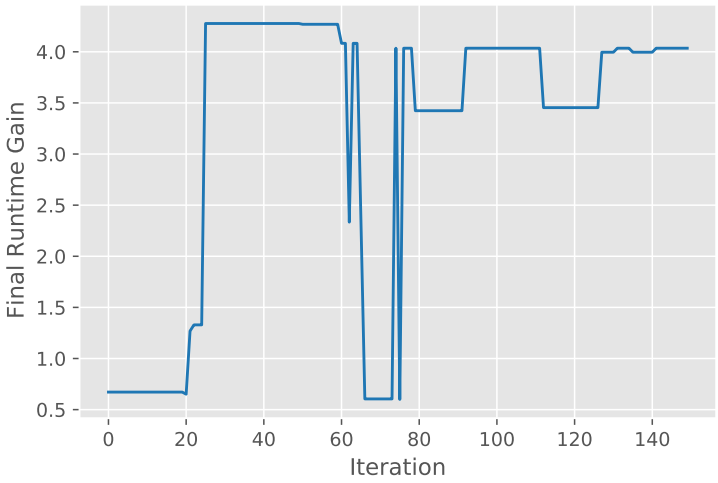
Figure 5.11: DQN generalization results for validation data.



(a) Episodic return per iteration.



(b) Final runtime for workload execution per iteration.



(c) Final runtime gain ratio w.r.t baseline.

Figure 5.12: DQN generalization results for test data.

Table 5.4: DQN Generalization statistics.

	Train	**Validation**	**Test**
Iteration			
Lowest return (cumulative)	0.0	0.0	0.0
Highest return (cumulative)	7.101011	7.101011	1.336995
Lowest final runtime (seconds)	5.409801	5.434819	3.245485
Highest final runtime (seconds)	440.981472	55.279418	23.139549
Final runtime gain (ratio)	0.786598	2.553981	4.276841

result is not reasonable. This also reflects during the testing period and the agent could receive around 5 times less reward than during training.

To extend our environment to include information about queries is one of our principal future works, and then we believe the agent will be able to generalize beyond the training query set.

By conducting this experiment, we tracked and solidify our hypothesis that the agent, as we have designed the environment, is not able to generalize successfully. To generalize beyond the training experiences, the agent requires to perceive information regarding the queries in the observation space.

In our next experiment, we continued our study of generalization with R2D2 agent. Since R2D2 is a memory based and advanced agent we would like to study how much variation we can see w.r.t generalizing.

5.3 Generalizing memory based discrete DRL agent (R2D2)

Experiment 4

In this experiment we will answer to our third research question:

1. To what degree is a modern memory-based discrete DRL algorithm (such as R2D2) able to generalize beyond the training experience for the partitioning use case?

Experiment set-up

The experimental setup we followed to answer this research question, concerning R2D2 [KOQ+18], is identical to our Experiment 3 set up. In terms of hyper-parameters, we used the same ones for the previous R2D2 experiment.

Results

In Figure 5.13 on the facing page, Figure 5.15 on page 89, and in Figure 5.16 on page 90 we report the result of training, validating and testing the agent R2D2 respectively. We also report the runtime gain ratio w.r.t the baseline in Figure 5.14 on page 88.

In Table 5.5 we report the statistics related generalization of DQN agent.

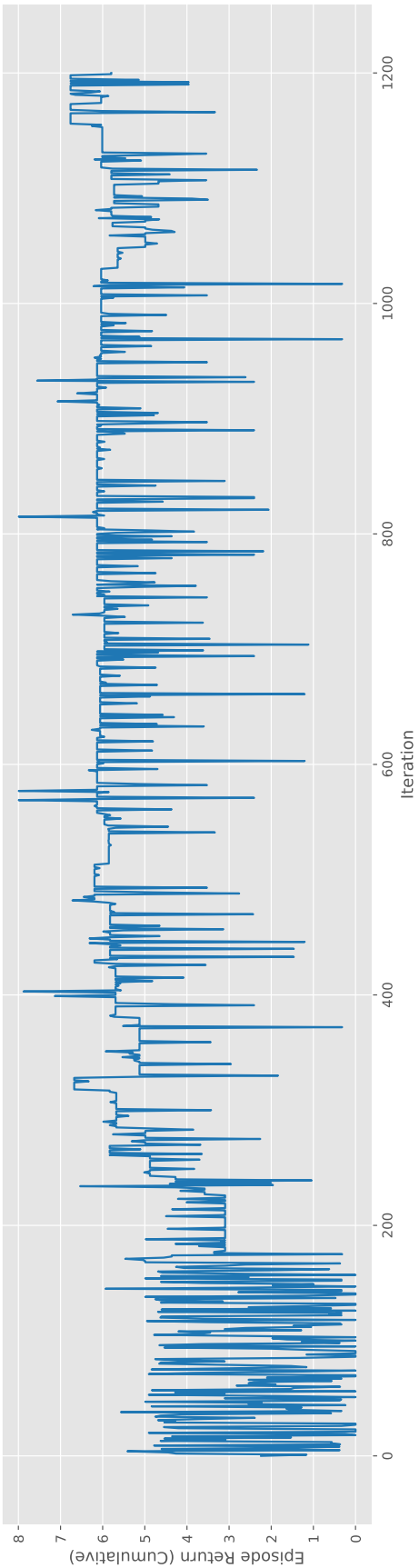
Table 5.5: R2D2 Generalization statistics.

	Train	**Validation**	**Test**
Iteration			
Lowest return (cumulative)	0.0	0.446558	0.613113
Highest return (cumulative)	7.99089	6.365145	3.27867
Lowest final runtime (seconds)	5.416923	5.726841	2.50149
Highest final runtime (seconds)	421.170396	192.745839	62.353775
Final runtime gain (ratio)	0.785564	2.423749	5.548863

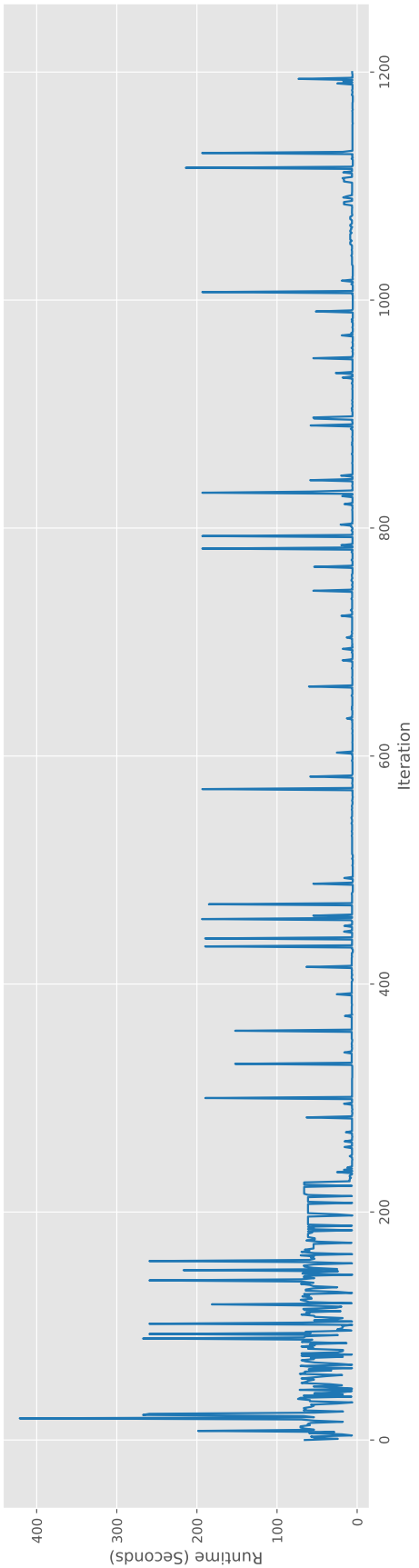
Discussion

In this experiment, during training we can observe a similar trend that we have seen before in our previous experiments both regarding episodic return and final runtime for executing the workload on the partitioned data. During training, the agent is not able to find a solution that is 0.785564 faster than the baseline. Even though, the highest reward achieved by the agent is 7.99089, which is the maximum compared to both validation and test scenarios, we believe that, as we have mentioned before, this metric is not suitable for judging the quality of the final partition.

In the validation scenario, we can observe that the agent is performing poorly and the model cannot stay stable w.r.t three metrics we have reported in Figure 5.15 on page 89. We see even worse result in the test scenario, the agent is completely unable to stay stable and all the metrics that we presented in Figure 5.16 on page 90 shows same chaotic behavior.

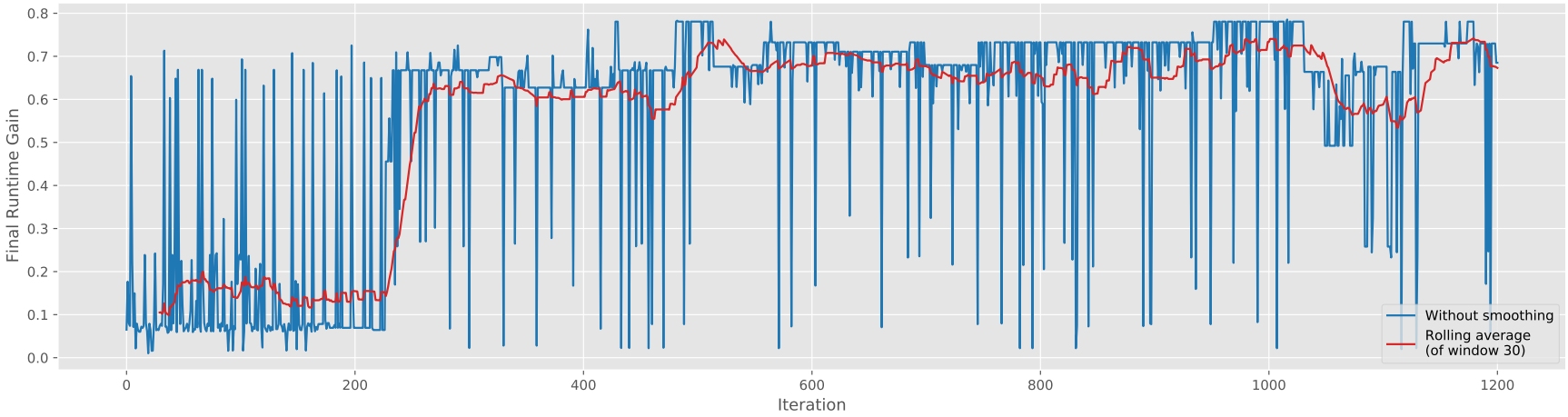


(a) Episodic return per iteration.



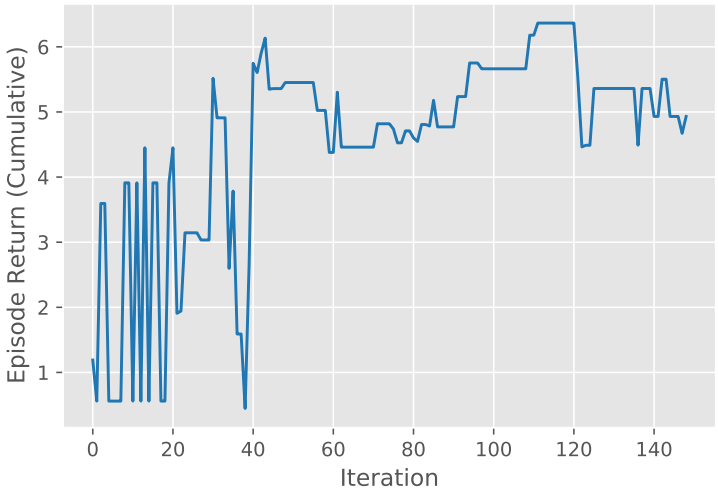
(b) Final runtime for workload execution per iteration.

Figure 5.13: R2D2 generalization results for train data w.r.t episodic return and final runtime.

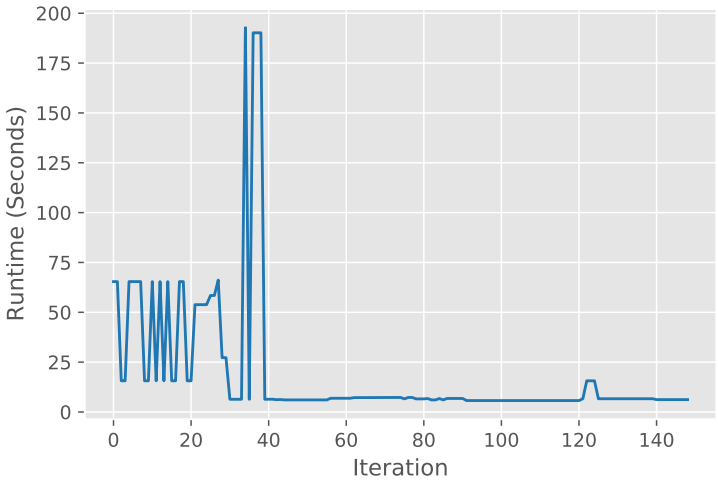


(a) Final runtime gain ratio w.r.t baseline.

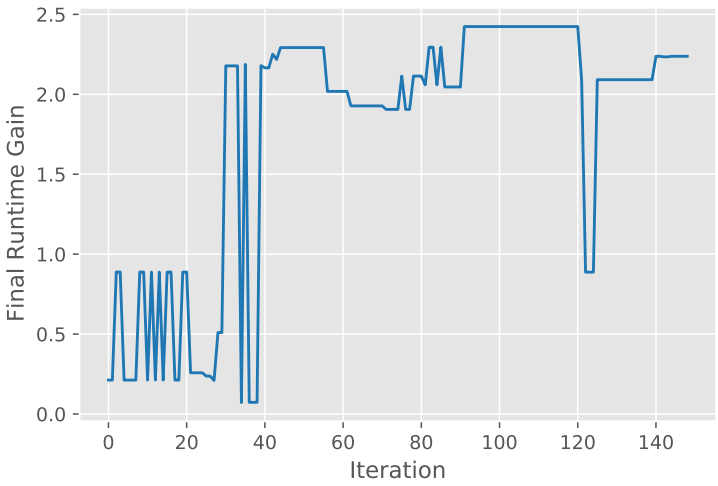
Figure 5.14: R2D2 generalization results for train data w.r.t final runtime gain.



(a) Episodic return per iterations.

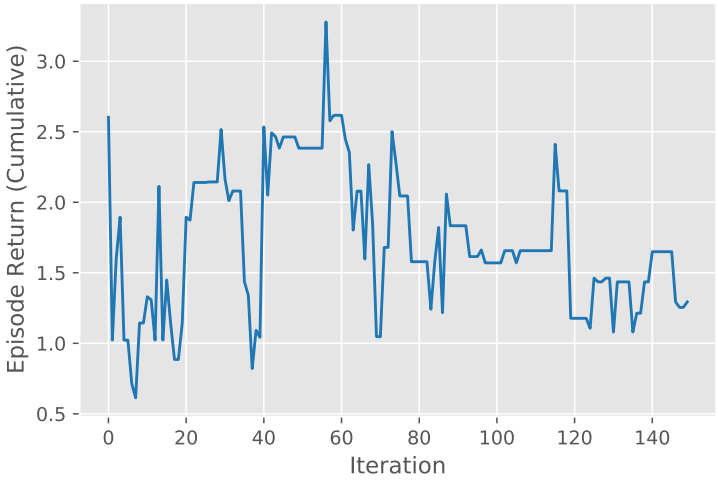


(b) Final runtime for workload execution per iteration.

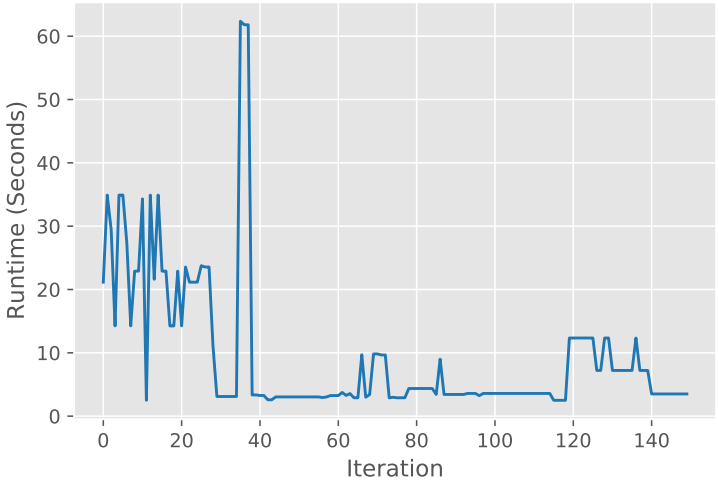


(c) Final runtime gain ratio w.r.t baseline.

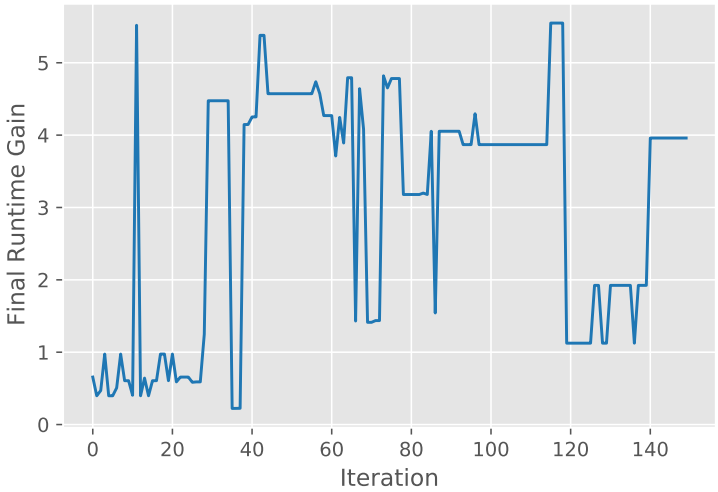
Figure 5.15: R2D2 generalization results for validation data.



(a) Episodic return per iteration.



(b) Final runtime for workload execution per iteration.



(c) Final runtime gain ratio w.r.t baseline.

Figure 5.16: R2D2 generalization results for test data.

From this experiment, we can hypothesize that without proper representation of the queries for the agent in the observation space, the agent is not capable to generalize beyond the training query set. For the agent to generalize beyond the training query set, the observation space that the agent receives from the environment after taking an action should contain the information concerning the query, for example, query frequency. As mentioned in Experiment 3, this is one of our main future works which we hope to continue in the future.

5.4 Summary

In this chapter, we have presented the experiments we have performed and the results. We also provided insights regarding the results we obtained along with possible limitations.

From our Experiments 1 and 2, we conclude that both DQN and R2D2 were able to find good horizontal partitions compared to the baseline, and performed around 1.7 times better for executing the workload w.r.t the baseline despite the large optimization space (i.e. the possible ways of partitioning the dataset) of 4654272. Still, we also observe that both agents perform surprisingly similar. We think this is due to the fact that our environment introduces too little steps for the R2D2 agent to bring improvements via its memory. We also find that agents are not able to converge to the optima that they find during training. This we believe is due to a need for more training.

From our Experiments 3 and 4, we reasoned that for the agent to be able to generalize beyond the training query set, the agent needs to perceive information regarding the queries in the observation space. Through Experiments 3 and 4, we solidified our intuition and tracked and reported the decline of the agents' performance. Still we observe that interestingly there are occasions while training where the agent finds competitive partitions on the testing case.

6. Conclusion and Future work

In this chapter, we are present a summary and conclusions we derived from our study. We also present potential directions concerning future work.

6.1 Conclusion

In this study, we have delved into the understanding of how deep reinforcement learning (DRL) can be adopted to utilize for data partitioning, with a focus on horizontal partitioning accompanying the use case of Spark [ZCF+10].

We have proposed three research questions, presented in Section 3.1 on page 41, to guide our study. Next, we introduced and implemented an environment based on the literature. We defined the complete functionality of the environment with the idea of making it extensible to distributed scenarios and other actions apart from partitioning(such as replication). We added improvements to our environment such as the offering of an action mask and caching of execution times for partitions. We also implemented two DRL agents using the Deepmind Acme framework: DQN [MKS+13, MKS+15] and R2D2 [KOQ+18]. Furthermore we studied the internal working of the algorithms to understand better their performance when using an action mask and we modified the algorithms to make a better use of the mask. With our implemented prototype encompassing agents, training loop, and environment, we created experiments to answer systematically to our research questions.

To answer to our first research question, we have conducted two experiments by employing DQN and R2D2. With these experiments, we have inferred that the DRL agents can obtain solutions better than the baseline concerning runtime for the execution of the workload. We find little difference in our case study between DQN and R2D2, we consider this to be a consequence of the limited number of steps in the task given to the agents, which fails to engage the memory mechanisms of R2D2 (we even did not use the burn-in feature). We also find that both agents do not converge to the exact optima identified during the training. We believe this to be due to the need for further tuning and training.

To answer to our second and third research questions, we have performed two additional experiments with DQN and R2D2. In our experiments, we have observed that the agents were not able to perform as we have considered earlier, and the agents could not generalize when we split some queries for training and some for testing. We hypothesize that this occurs due to the absence of query information in the observation space. This eventually impacts the generalization capability of the agent, considering that the agent is unable to perceive any information from the environment associated with the queries. We also hypothesize that this limitation is entirely a consequence of the design of our environment, where the information regarding the queries is not available. In future work we seek to improve this.

Apart from the aforementioned conclusion from our study, we have provided an environment for data partitioning that can be utilized by others for partitioning any database using DRL agents while relying on SparkSQL and parquet files. We have also considered caching strategies concerning workload execution runtime, and partitioned data for the environment, which accelerated the training process. Furthermore, we incorporated an action mask for the agents as an extension to the original models, which contributes to their training stability. Moreover, we also contribute early results, and analysis for horizontal data partitioning by applying two DRL agents from the standard DRL framework, ACME [HSA+20].

6.2 Future work

Following, we provide a list of possible future works based on this study.

1. Even though we have adopted Spark [ZCF+10] in our work, we have not taken full advantage of a distributed system and have worked with a single machine. In the future, we would like to extend this work towards a distributed system. We also worked with a very small database of TPC-H. In the next work we should scale to larger databases, and if possible, study the impact of training on samples.

2. One of our principal future works is to act towards making generalization achievable. In this study, we have pointed out the limitation we had in the design of our environment, in the future we would want to incorporate more functionality to our environment including query information in the observation space. Which we believe will be the proper step towards generalization.

3. In our study we have considered the execution time on non-partitioned data as our baseline. This was helpful to know the improvements that our agents bring to the process. However we cannot be sure if our agents are far or close to the best improvements possible. We think that if we could determine the optimal partitioning based on dynamic programming we would be able to understand this better. Unfortunately this is not trivial since we do not have a cost model and instead are relying only on execution times, which could slow down very much the dynamic programming search for the optima. In future work we think it is necessary to solve this aspect so we can understand better the performance of our models for the application.

4. For reward calculation in this study, we have worked with Pythons time function which we consider not to be precise enough to represent the true workload execution runtime since it considers other time while reporting elapsed time (for example query parsing, I/O, and others). For calculating the execution time for the workload, and excluding any other time that might be involved throughout the processing of the query (for example, parsing query text) and other I/O times, in the future, we would like to adopt a more domain-specific library which is SparkMeasure [Can19].

5. For reward calculation we also believe that the use of caching of query runtimes could be a cause for errors, as perhaps on the early stages when there is a lot of re-partitioning the query runtime might be affected by that, while in later stages the less number of partition creation might lead to faster execution. Hence, we think that some revision to double-check the stability of the caching is important.

6. In this study we have partitioned the database horizontally without considering replication, in the future, we would like to incorporate replication of a database in our approach too to study its effect on workload execution runtime

7. In our experiment, we have introduced three stopping conditions that will lead an episode to terminate and reset to its initial state. One of the stopping conditions was associated with: if a certain action takes more than a threshold time then the current episode will be terminated and reset to the initial state. After conducting several experiments we have found that working with time-out is not a trivial approach to associate with our environment. To mitigate this issue we have set a high time-out value, which has worked suitably with our experiments. We hypothesize that for our environment to work, the time-out concept needs to be revised as we have it now, or else the environment might not perform as intended. In the future, we would work to improve our time-out notion and make it work with our environment.

8. In our environment design we have made the agents perform a fixed number of partitioning steps. In future work, we would like to add a "quit" action, which would enable agents to perform more or less partitionings than a number.

9. In our environment design we have not considered any logic to make agents pick the most competitive partitioning first. In future work we would like to consider this addition, as we believe it could facilitate the training process since agents could be able to have a precise order of actions that bring the highest reward. In our case this is not implemented, since the rewards for a set of actions would be equivalent irrespective of the order of the actions.

10. We have only used the TPC-H[14] benchmark dataset for our study and we wish to use other benchmark datasets in the future to be more representative of the real-world database.

[14]http://www.tpc.org/tpch/

11. In this study, we have tried to mimic (partially) the work of Hilprecht [HBR20], but in the future, we would like to mimic and work with the approach based on a QD-tree and query routing[YCW+20].

12. In the future we would like to work further with the hyperparameter configuration of the two agents we have worked with within our study, to determine which set of hyperparameters is suitable for solving the data partitioning problem.

13. We have used two agents in our study DQN and R2D2, we would want to extend our research by adopting more advanced agents, for example, Agent57 [BPK+20], which has been capable to successfully perform better than human in all the games of Atari 57 [BNVB13]. Similarly we would like to attempt agents able to perform multi-objective optimization, and agents able to learn from demonstration.

14. For our study we have focused on the rewards obtained by the agents, while considering the models of the agents as black boxes. In future work we would like to add some degree of interpretability to our framework. We would like, for example, to understand how query access patterns relate to optimal partitions and how this is perceived by the model.

Bibliography

[ANY04] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370, 2004.

[BLT+16] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab, 2016.

[BNVB13] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[BPK+20] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark, 2020.

[Can19] Luca Canali. sparkmeasure. https://github.com/LucaCanali/sparkMeasure, 2019.

[CN07] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14, 2007.

[DC04] Adrian Sergiu Darabant and Alina Campan. Semi-supervised learning techniques: k-means clustering in oodb fragmentation. In *Second IEEE International Conference on Computational Cybernetics, 2004. ICCC 2004.*, pages 333–338. IEEE, 2004.

[DD11] Adrian Sergiu Darabant and Laura Darabant. Clustering methods in data fragmentation. *Rom. Journ. of Information Science and Technology*, 14(1):81–97, 2011.

[DHYD20] Zihan Ding, Yanhua Huang, Hang Yuan, and Hao Dong. Introduction to reinforcement learning. In Hao Dong, Zihan Ding, and Shanghang Zhang, editors, *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pages 47–123. Springer Singapore, Singapore, 2020.

[DPP+18] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S Sekeran, Fabián Rodriguez, and Laxmi Balami. Gridformation: towards self-driven online data partitioning using reinforcement learning. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–7, 2018.

[DPP+19] Gabriel Campero Durand, Rufat Piriyev, Marcus Pinnecke, David Broneske, Balasubramanian Gurumurthy, and Gunter Saake. Automated vertical partitioning with deep reinforcement learning. In *European Conference on Advances in Databases and Information Systems*, pages 126–134. Springer, 2019.

[Dur] Gabriel Campero Durand. Production-ready learning-augmented data management with deep reinforcement learning.

[ECN15] Cheikh Kacfah Emani, Nadine Cullot, and Christophe Nicolle. Understandable big data: a survey. *Computer science review*, 17:70–81, 2015.

[FH19] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Automated Machine Learning: Methods, Systems, Challenges*, pages 3–33. Springer International Publishing, Cham, 2019.

[FKN11] Florian Funke, Alfons Kemper, and Thomas Neumann. Benchmarking hybrid oltp&olap database systems. *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2011.

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

[HBR20] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Learning a partitioning advisor for cloud databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 143–157, 2020.

[HN79] Michael Hammer and Bahram Niamir. A heuristic approach to attribute partitioning. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 93–101, 1979.

[HQB+18] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay, 2018.

[HR15] Sandhya Harikumar and Raji Ramachandran. Hybridized fragmentation of very large databases using clustering. In *2015 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*, pages 1–5. IEEE, 2015.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HS17] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps, 2017.

[HSA+20] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

[KOQ+18] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[Li18] Yuxi Li. Deep reinforcement learning. *arXiv preprint arXiv:1810.06339*, 2018.

[Lin93] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[LLPS93] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[MDMS69] WT McCormick, SB Deutsch, JJ Martin, and PJ Schweitzer. Identification of data structures and relationships by matrix reordering techniques (tr p-512). *Institute for Defense Analyses, Arlington, VA*, 1969.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997.

[MJSW72] William T McCormick Jr, Paul J Schweitzer, and Thomas W White. Problem decomposition and data reorganization by a clustering technique. *Operations research*, 20(5):993–1009, 1972.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[OOC07] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb), 2007.

[PAA+17] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving database management systems. In *CIDR*, volume 4, page 1, 2017.

[RMAHAFPC14] Lisbeth Rodríguez-Mazahua, Giner Alor-Hernández, Ma. Antonieta Abud-Figueroa, and S. Gustavo Peláez-Camarena. Horizontal partitioning of multimedia databases using hierarchical agglomerative clustering. In Alexander Gelbukh, Félix Castro Espinoza, and Sofía N. Galicia-Haro, editors, *Nature-Inspired Computation and Machine Learning*, pages 296–309, Cham, 2014. Springer International Publishing.

[RZML02] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 558–569, 2002.

[SB18] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction 2nd ed, 2018.

[SFKX14] Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1115–1126, 2014.

[Sil15] David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.

[SQAS16] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[SSD18] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643*, 2018.

[SW85] Domenico Sacca and Gio Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems (TODS)*, 10(1):29–56, 1985.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[Wat89] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge United Kingdom, 1989.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[WSH+16] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[YCW+20] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 193–208, 2020.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[zV11] M. Tamer zsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2011.