Otto-von-Guericke-University Magdeburg

Faculty of Computer Science
Department of Databases and Software Engineering

A Comparative Evaluation of Deep Reinforcement Learning Frameworks

Master Thesis

Author:
Pavlo Shevchenko

Examiner and Supervisor:
Prof. Dr. rer. nat. habil. Gunter Saake

2nd Examiner:
Dr.-Ing. Christian Rössl

Supervisor:
MSc. Gabriel Campero Durand

Magdeburg, 13.09.2019
Pavlo Shevchenko
A Comparative Evaluation of Deep Reinforcement Learning Frameworks
Abstract

In recent years, *deep reinforcement learning* (DRL) has emerged as a powerful and general approach for a variety of sequential decision making tasks. While early research has been mainly limited to controlled environments like complex video games and scoped robotic tasks, recent works have shown large potential in applying DRL approaches for everyday tasks in computer systems. A large number of frameworks providing reference implementations of common deep reinforcement learning algorithms makes it possible nowadays for the researchers and developers to quickly prototype new ideas for DRL applications. However, due to the heterogeneous nature of these solutions and absence of formal guidelines, choosing a framework that fits the particular use case of a developer and some given system requirements may be challenging.

Therefore, in this research we conduct the first comparative evaluation of several state-of-the-art DRL frameworks in an attempt to improve researchers’ and developers’ awareness of a wide range of existing solutions, and how they compare. In particular, in this work, we provide an evaluation platform, using it to analyze key components and features of such representative frameworks as the research-oriented and lightweight *Dopamine* library, the end-to-end *Horizon* platform for large-scale production systems, and the *Ray RLlib* framework, which is optimized to enhance learning in a distributed setting. By varying numerous parameters in the frameworks’ configurations, we simulate several scenarios and report framework performance both in terms of runtime and in the stability of the learning process.

Empirical results obtained for a traditional *CartPole* and real-world *Query Optimizer* environments indicate that Ray RLlib’s parallel abstract components provide the best performance both on CPU and GPU. Although Horizon agents failed to achieve the optimal performance for the chosen common configuration and network architecture, this framework offers numerous features valuable in a production setting where access to the environment might be restricted. Dopamine, on the other hand, managed to provide both a compact implementation and satisfying performance across a number of experiments.

While by no means exhaustive, our research provides a comprehensible overview of existing DRL solutions and highlights the main differences in their design and purposes. By sharing our flexible evaluation platform, we encourage the scientific community to contribute to our study by implementing new experiments for already integrated frameworks and other promising solutions.
Acknowledgements

First and foremost, I would like to express my sincere gratitude to Gabriel Campero Durand for introducing me to the world of deep reinforcement learning and keeping me motivated throughout this work.

I would also like to thank Prof. Dr. rer. nat. habil. Gunter Saake for making this thesis at the DBSE research group possible, and Dr.-Ing. Christian Rössl for not only providing his valuable feedback during this work, but also for supporting me on the early stages of my academic career.

Besides my supervisors, I thank HPI Future SOC Lab Team for letting me use their computation resources and their help with setting up a robust evaluation environment.

Finally, I would like to sincerely thank my mother for helping me overcome discouraging phases of this work and always supporting me in my pursuit of happiness.
Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

________________________________________  ______________________________________
Signature                                      Place, Date
# Contents

List of Figures 3

1 Introduction 5
   1.1 Motivation 5
   1.2 Main Contributions 6
   1.3 Thesis Structure 7

2 Background 9
   2.1 Reinforcement Learning 9
      2.1.1 Formal Definition 9
      2.1.2 Different Learning Settings and Strategies 11
      2.1.3 Deep Reinforcement Learning 12
   2.2 DRL Frameworks 19
      2.2.1 Dopamine 19
      2.2.2 Horizon 20
      2.2.3 Ray RLlib 24
      2.2.4 Landscape of DRL Frameworks 27
   2.3 Benchmark Environments 28
      2.3.1 OpenAI Gym 28
      2.3.2 Park Project 29
      2.3.3 AI Safety Gridworlds 31
   2.4 Summary 32

3 Design of the Evaluation Platform 35
   3.1 Research Questions 35
   3.2 Design of the Evaluation Platform 35

4 Experimental Setup 39
   4.1 Benchmarks 39
      4.1.1 OpenAI Gym’s CartPole Environment 39
      4.1.2 Park’s Query Optimizer Environment 40
      4.1.3 Join Order Benchmark 42
   4.2 Configuration of DRL Agents 43
   4.3 Evaluation Environment 45
   4.4 Summary 46

5 Evaluation and Results 47
   5.1 Complete Suite of Conducted Experiments 47
   5.2 Evaluation Results for OpenAI Gym’s CartPole Environment 48
   5.3 Evaluation Results for Park’s Query Optimizer Environment 54
   5.4 Summary 58
# Contents

6 Related Work 59

7 Conclusion and Future Work 61

7.1 Conclusion .................................................. 61
7.2 Threats to Validity ............................................. 62
7.3 Future Work .................................................. 63

Bibliography 67

Appendices 75

A. Example Configurations of the DQN Agent for the CartPole Environment. 75
B. Example Configurations of the DQN Agent for the Query Optimizer Environment. 78
C. Unsmoothed Query Optimizer Rewards Observed by Evaluated DRL Agents. 81
List of Figures

2.1 Agent-environment interaction loop in reinforcement learning (adapted from [FLHI+18]). .......................................................... 10
2.2 A non-exhaustive taxonomy of state-of-the-art reinforcement learning agents1. 12
2.3 A schematic representation of a deep Q-network. ........................... 13
2.4 Network architectures for DQN and recent distributional RL agents (original image by [DOSM18]). ........................................... 16
2.5 The practice-oriented architecture of a PPO agent. .......................... 18
2.6 Architecture of Dopamine framework. Blue boxes represent software components, while yellow boxes indicate component’s files and corresponding lines of code (original image by [CMG+18]). .......................... 19
2.7 Horizon’s DRL pipeline for production environments2. ....................... 21
2.8 A high-level representation of Horizon’s production feedback loop3. ....... 23
2.9 Overview of common components in popular deep reinforcement learning agents (original image by [LLM+17]). ........................... 24
2.10 An overview of state-of-the-art distributed control models (adapted from [LLM+17]). Red dotted edges denote the control flow, while dark blue solid edges indicate data flow. ............................................. 25
2.11 Architecture of the task-based programming model used in Ray (original image by [MNW+18]). ................................................. 25
2.12 Park’s request-response architecture (adapted from [MNN+19]). ............. 29
2.13 Overview of computer system environments supported by Park platform (original image by [MNN+19]). ...................................... 30
2.14 The absent supervisor environment (original image by [LMK+17]). ........... 31
3.1 Design of our platform for evaluation of deep reinforcement learning frameworks. ............................................................................. 36
3.2 Overview of extensions done in our work to the frameworks’ standard components. ............................................................................ 37
4.1 A schematic representation of cart-pole control task (adapted from [BSA83]). 40
4.2 Schematic representation of a model with action pruning. ...................... 41
4.3 A query graph corresponding to query 1a of the Join Order Benchmark (adapted from [LGM+15]). ......................................................... 42
4.4 Unified neural network for evaluated deep reinforcement learning agents. 43
5.1 Average rewards per evaluation episode (left) and the total duration of the learning process (right) for the frameworks’ implementations of a DQN agent and OpenAI Gym’s CartPole environment. ............................ 49
5.2 Profiling results for the first five iterations of the agent’s interaction with the CartPole environment. Left chart shows raw runtimes, while the right chart presents the percentage of runtime spent performing individual phases. 50
5.3 Total runtime of the learning process with varying checkpointing frequency (left), and memory and time consumption of the checkpointing step (right) for the CartPole environment. ............................................. 51
5.4 Average rewards per evaluation episode (right) and the total duration of the learning process (left) for the frameworks’ implementations of several DRL agents and OpenAI Gym’s CartPole environment. .......................... 52
5.5 Average time required by the agent to perform an action in the CartPole environment. ................................................................. 53
5.6 Average rewards per evaluation episode (left) and the total duration of the learning process (right) for the frameworks’ implementations of the DQN agent and Park’s Query Optimizer environment. .................. 54
5.7 Smoothed average rewards obtained for the Query Optimizer environment. 55
5.8 Profiling results for the first five iterations of the agent’s interaction with the Query Optimizer environment. Left chart shows raw runtimes, while the right chart presents the percentage of runtime spent performing individual phases. ................................................................. 55
5.9 Smoothed average rewards per evaluation episode (right) and the total duration of the learning process (left) for the frameworks’ implementations of several DRL agents and Park’s Query Optimizer environment. ........ 56
5.10 Average time required by the agent to perform an action in the Query Optimizer environment. ....................................................... 57
1 Introduction

We start this thesis with an overview of recent developments in the field of machine learning that inspired our research. In Section 1.1, we review some of the most influential works in the field of reinforcement learning. This review serves to define the scope of our work. In Section 1.2, we highlight the main contributions of our research, and conclude this chapter with the structure of subsequent sections in Section 1.3.

1.1 Motivation

Learning by trying and failing is the core principle of how humans obtain new knowledge and skills. Throughout our lives, we continuously make decisions, observe the consequences that they have on the world around us and we adjust our behavior to achieve certain goals. Reinforcement learning (RL), proposed by Sutton et al. [Sut88], provides a framework to formalize this type of learning for artificial actors that deal with sequential decision-making tasks. The main idea is that an artificial agent uses past experiences of interaction with an uncertain environment to develop an optimal strategy that yields maximal future benefits. RL’s flexible definition makes it applicable to a wide range of problems with deterministic or stochastic environments, low- and high-dimensional action spaces, and fully or partially observable situations [FLHI+18].

Over the past few years, deep reinforcement learning (DRL) approaches, which extend RL and are based on deep neural networks, have been shown to be able to achieve superhuman performance for a variety of complex tasks like playing video games [MKS+13] or robotic manipulation [BCP+16]. While early successes in the field of deep reinforcement learning were mainly limited to traditional simulated environments, recent research has shown high interest in applying DRL to real-world sequential decision making tasks. Examples of recent applications include scheduling jobs on distributed clusters [MSV+18], internet congestion control [JRG+18] and SQL query optimization [KYG+18], among many others. To encourage further algorithmic research for production-related problems, the recently introduced Park benchmark provides a suite of 12 representative computer system environments that pose new unique challenges to the scientific community [MNN+19].

Although recent research has shown obvious benefits of using DRL techniques in industrial tasks, very few commercial applications have actually integrated such agents in their products [GCL+18]. Numerous existing off-the-shelf solutions are designed to assist both researchers and developers in applying these novel approaches to their particular use cases [FL+16][CMG+18][GCL+18][CLNE17]. However, in the absence of comparative evaluations of these frameworks, picking the right solution for a specific problem becomes a non-trivial task. In this research, we make the first attempt to analyse a wide landscape
of DRL frameworks and provide developers with some guidelines to boost the industrial adoption of the cutting-edge research developments.

Due to time and resource limitations on this research, we are not able to provide an exhaustive comparison of all existing frameworks. Therefore, in this work we focus on three state-of-the-art solutions that, we judge, represent the complete spectrum of existing frameworks. Firstly, we have a look at a research-oriented Dopamine framework [CMG+18] that provides a lightweight and easily extendable implementation of several state-of-the-art value-based DRL agents. Secondly, we review the features provided by the Horizon platform [GCL+18] designed for an end-to-end support of DRL-based production applications. Finally, we analyze the novel design of the Ray RLlib framework [LLM+17] that uses parallel abstract components to implement highly-distributed DRL agents.

We acknowledge the fact that an exhaustive evaluation of such software products requires thorough analysis of numerous solution’s aspects like quality, size and maintainability of the code base, framework’s memory footprint, robustness, etc. However, to keep our study concise and focused, we evaluate only the performance aspects of implementations provided by the frameworks. By varying a number of parameters in the configuration of selected solutions, we are able to simulate several production situations and provide recommendations of the implementations that are both fast and achieve stable learning behavior in such scenarios.

Herewith, we have outlined the scope of our research; the main contributions of this thesis are summarized in the following section.

### 1.2 Main Contributions

1. We review a wide variety of existing frameworks that provide essential DRL functionality, while the majority of the work is dedicated to three frameworks: Dopamine, Horizon and Ray RLlib. In particular, we present the main features and technology stack of these frameworks, and discuss framework’s key components responsible for its performance.

2. We present our design of a flexible evaluation platform that employs best development practices to ensure a better reproducibility of experimental results. By making our platform publicly available\(^1\), we encourage community’s involvement in integration of new frameworks and experiments.

3. We conduct a series of experiments on two commonly used benchmarks: OpenAI Gym [BCP+16] and Park [MNN+19], and report the performance of selected frameworks’ agents both in terms of runtime and convergence of the learning process.

4. Based on these empirical results, we indicate the most efficient implementations for specific scenarios, and identify some of the frameworks’ weaknesses that have to be considered when integrating these solutions in production systems.

\(^1\)[https://github.com/pshevche/drl-frameworks]
5. We outline some promising directions for future research that should improve our understanding of existing DRL solutions and enhance their introduction in real-world applications.

1.3 Thesis Structure

The rest of the thesis is structured as follows:

- In Chapter 2, we explain some fundamental concepts crucial for the deep understanding of the discussed problem. In particular, we revisit the theory behind deep reinforcement learning, and shed some light on the wide landscape of DRL frameworks.

- Chapter 3 formulates the goals of this thesis in more specific research questions, and introduces our view of a platform for evaluation of deep reinforcement learning frameworks.

- In Chapter 4, we review such essential components of the experimental setup like benchmark data selected, algorithms’ configurations and specifications of utilized resources.

- Chapter 5 is dedicated to the review and discussion of evaluation results obtained for two selected benchmark environments and varying hyper parameters.

- In Chapter 6, we revisit some previous research related to the evaluation of machine learning frameworks and applying deep reinforcement learning concepts in production setting.

- Finally, we conclude this thesis and outline some intriguing directions for future research in Chapter 7.
2 Background

In the following, we provide theoretical background relevant to the experimental setup and evaluation results presented in the later sections. The rest of the section is organized as follows:

- In Section 2.1, we briefly discuss the ideas behind (deep) reinforcement learning. In particular, the formal definition of RL tasks, as well as the workflow of several DRL algorithms, are explained.
- Section 2.2 is dedicated to the overview of a wide range of off-the-shelf DRL frameworks. We start with an exhaustive description of the architecture of evaluated frameworks, and conclude by reviewing some promising solutions not included in this study.
- Finally, in Section 2.3, we revisit some of the traditional and recently introduced environments used as benchmarks for evaluation of deep reinforcement learning concepts. In particular, we discuss OpenAI Gym and Park benchmarks which were used in our evaluation.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning (ML) that deals with tasks, where a decision maker has to take a sequence of actions that should result in maximal future rewards [FLHI18]. A key aspect of reinforcement learning is that an actor doesn’t possess a complete knowledge of the problem, but learns through trial-and-error while acquiring more information about the task [SB18]. In the following, we briefly revisit the formal definition of such tasks and present several methods. We especially focus on those reinforcement learning algorithms that use deep neural networks and which are used in our evaluation.

2.1.1 Formal Definition

Traditionally, sequential decision making problems are defined using finite Markov Decision Processes (MDP) [Bel57]. An MDP is a discrete time stochastic control process defined as a 5-tuple \((S, A, T, R, \gamma)\) where \(S\) is the state space, \(A\) is the action space, \(T : S \times A \times S \rightarrow [0, 1]\) is the transition function describing transitional probability between states, \(R : S \times A \times S \rightarrow [0, R_{\text{max}}]\) is the reward function and \(\gamma \in [0, 1)\) is the discount factor. The key property of a Markov Decision Process is that the future of the process only depends on the current observation and all past observations are irrelevant [FLHI18]. As a result, MDPs perfectly represent sequential decision making tasks where chosen actions...
affect not only intermediate rewards, but also future states of the problem and future rewards [SB18]. The learner of such MDP problem is called the *agent*, whereas the problem it interacts with is known as the *environment*. The agent interacts with the environment in a continuous loop, depicted in Figure 2.1.

**Figure 2.1:** Agent-environment interaction loop in reinforcement learning (adapted from [FLHI+18]).

At each discrete timestep, the agent observes the current *state* of the environment and picks the best possible *action* in this situation. At the next step, the agent receives a response from the environment containing the new state of the problem and a numerical *reward* for choosing this action.

In addition to the agent, the environment and the reward signal, a reinforcement learning system contains the following key components:

- **Policy** $\pi$ defines the agent’s strategy for picking actions given the current environment’s state, which can be deterministic ($\pi(S_t) = A_t$) or stochastic ($\pi(S_t, A_t) = P_t$). Generally speaking, the main goal of the RL agent is to find an optimal policy $\pi^*$ which guarantees maximal rewards in the long run.

- While rewards help us assess the quality of intermediate decisions, a *value function* can estimate the long-term benefits of choosing an action [SB18]. A value of a state is the total amount of rewards an agent can expect in the future by starting in this state and following the current policy:

$$V^\pi(s) = \mathbb{E}\left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, \pi \right]$$

(2.1)

Based on this definition, we can refine the agent’s goal to finding a policy that yields the optimal/maximal expected return.

- In addition to the aforementioned value function, some approaches use the notion of the *quality function* in their computations. Similar to the value function, the

\[1\] Formally, a discrete time stochastic control process has the Markovian property if:

- $P(\omega_{t+1} \mid \omega_t, a_t) = P(\omega_{t+1} \mid \omega_t, a_t, \ldots, \omega_0, a_0)$
- $P(r_t \mid \omega_t, a_t) = P(r_t \mid \omega_t, a_t, \ldots, \omega_0, a_0)$

, where $\omega_t$ is a representation of environment’s state at timestep $t$, $a_t$ is the agent’s action at timestep $t$ and $r$ is the reward for choosing action $a_t$.  

1
quality function is defined as:

\[ Q^\pi(s, a) = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a, \pi] \] (2.2)

Informally, the quality function estimates the long-term reward after choosing action \( a \) in state \( s \). The advantage of using the quality function instead of the value function is that the optimal policy can be obtained directly from the optimal \( Q \)-value function [FLHI+18]:

\[ \pi^*(s) = \arg\max_a Q^*(s, a) \] (2.3)

Therefore, some reinforcement learning algorithms aim to find the optimal \( Q \)-value function, which subsequently allows us to derive the optimal policy. By iteratively approximating the quality function for the environment, the agent is incrementally reaching the optimal policy. The family of approaches that use this concept are called value-based algorithms. \( Q \)-learning introduced by Watkins et al. [WD92] is one of the original methods.

- In order to predict the behavior of the environment, some agents may learn the model of the environment. This is some abstraction that allows to predict future states and rewards based on the current state and action. However, such approaches fall outside the scope of our research.

To sum it up, reinforcement learning tasks are usually formalized using Markov Decision Processes. The decision maker (called the agent) interacts with the problem (called the environment) in a continuous loop. At each step, the agent observes the state of the environment and tries to choose an action that yields maximal future rewards. Several empirical measures like value and quality functions can be used to estimate these expected returns. The optimal quality function for the environment allows us to directly derive the agent’s policy, a strategy for picking actions based on the observed state. Approaches that use this property are called value-based techniques, and \( Q \)-learning is one of them. Finally, to improve the sample efficiency, some agents may learn the model of the problem which helps them predict future states of the environment [FLHI+18].

2.1.2 Different Learning Settings and Strategies

Reinforcement learning approaches can operate in different settings when collecting new experiences. Firstly, in offline setting, the agent is provided with limited data and has no ability to interact with the environment to acquire new experiences [SB18]. Contrarily, in online setting, the agent, in parallel to learning, has the ability to gather new observations from the environment and store them for later use [FLHI+18]. Although the core reinforcement learning algorithms for offline and online learning are basically the same, they have to deal with different challenges depending on the setting. Offline learning is closely related to the concept of generalization. Despite a rather restricted knowledge about the environment, the agent should efficiently use provided experiences and find a policy that will perform well on the fully exposed environment or a similar one [KLM96]. In case of online learning, the agent has to solve the so-called exploration vs exploitation dilemma. Although it is desired that the agent quickly discovers and converges against the optimal policy (exploitation), it should also investigate sub-optimal policies.
in the earlier stages of learning (exploration) [Thr92]. Exploring the complete observation space is especially important in the environments where consecutive observations are very similar (e.g. video frames) and can force the agent into using sub-optimal policies [MKS+15].

In the online setting, one can differentiate between RL agents that perform on-policy and off-policy learning. The former attempts to incrementally improve the policy used to interact with the environment and collect new experiences [SJLS00]. The latter, on the other hand, uses two different policies: one for gathering new observations (called behavior policy) and one that is being learned and eventually becomes the optimal policy (also known as target policy) [SB18]. While the on-policy methods are considered to be safer [SJLS00] and more robust [FMP18], the off-policy approaches are more sample efficient [FLHI+18] and general [SB18]. Since off-policy approaches can learn from any policy, observed experiences can be stored in a replay buffer and be re-used during numerous learning phases. This also makes such techniques applicable in offline setting described previously [SB18].

In the following, we’ll review some of the on-policy and off-policy reinforcement learning methods used in the evaluation part of this research.

![Figure 2.2: A non-exhaustive taxonomy of state-of-the-art reinforcement learning agents](https://spinningup.openai.com)

### 2.1.3 Deep Reinforcement Learning

Real-world sequential decision making tasks tend to have high-dimensional observation spaces and often continuous action spaces [GCL+18]. Traditional value-based RL algorithms, however, are proven to be efficient only for rather small discrete state and action spaces [MKS+15]. Over the past few years, several new reinforcement learning agents were proposed that attempt to solve these issues. These new approaches, referred to as

---

Deep Reinforcement Learning (DRL) agents, use deep neural networks\(^3\) to approximate the value function and derive the optimal policy [FLHI\(^+\)18]. By using deep neural networks, these agents are able to successfully solve complicated tasks with very little prior information about the environment.

In the following, we briefly discuss several DRL agents used in the evaluation part of this research. It is worth mentioning that we don’t attempt to provide an exhaustive explanation of agents’ components, but only outline the key differences between the approaches and provide references for further reading. We’ll start by revisiting the concept behind the original DRL agent, called Deep Q-Network (DQN) [MKS\(^+\)15], then we review two approaches built on top of DQN, namely the Rainbow Agent [HMVH\(^+\)18] and Implicit Quantile Networks (IQN) [DOSM18], and conclude this section with an overview of the Proximal Policy Optimization agent [SWD\(^+\)17], a state-of-the-art on-policy RL agent. Of course, the family of deep reinforcement learning agents is not limited to these algorithms; a broader but still incomplete overview of the latest DRL agents is depicted in Figure 2.2.

**Deep Q-Network**

As mentioned previously, Deep Q-Network introduced by Mnih et al. [MKS\(^+\)13] is the first RL agent that uses deep neural networks to approximate the quality function. This novel approach was able to achieve super-human level for a variety of Atari 2600 video games by learning the policy from a sequence of video frames [MKS\(^+\)15].

![Figure 2.3: A schematic representation of a deep Q-network.](image)

The network schematically depicted in Figure 2.3 takes the state of the environment as an input (e.g. video frames), feeds it to the deep neural network\(^4\) and returns an approximate Q-value for each of the possible actions (e.g. button to press). The parameters of the underlying network are optimized by using stochastic gradient descent to minimize the loss:

\[
L = \left( r + \gamma \max_{a'} Q(s', a') - \frac{Q(s, a)}{\text{prediction}} \right)^2
\]  \hspace{1cm} (2.4)

\(^3\)Neural networks are a set of algorithms, designed to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates (Definition by https://www.investopedia.com/terms/n/neuralnetwork.asp).

\(^4\)The original paper employed a convolutional neural network with three convolutional and two fully-connected layers.
Previous research has shown that reinforcement learning tends to be unstable or even diverge if a neural network is used to approximate the quality function [TVR96]. DQN solves this issue by employing the following two ideas in its design:

- **Experience replay.** During the interaction with the environment, the agent stores collected experiences in a replay buffer. Later, during the training phase, random samples from the replay buffer are used instead of the most recent observations. Choosing random transitions breaks correlations in subsequent observations, which otherwise would drive the agent into a local minimum.

- **Target network.** In a traditional setting, a single network is used to set the target for learning and to make predictions for the current state. This causes the problem that by updating the weights of the network according to the computed loss, we also update the target. This results in a moving target which has a negative impact on the convergence of the learning method. To tackle this issue, DQN uses two different networks: one that sets the targets and is updated every $C$ steps, and one that makes the predictions and is updated after each step. Once $C$ steps elapsed, both networks will be synchronized and new targets will be set. This strategy fixes the target values for several iterations which results in a better convergence of the method.

All these concepts put together result in the following algorithm for Deep Q-Learning:

**Algorithm 1:** Deep Q-Learning with experience replay and target network (adapted from [MKS+15])

```plaintext
1. Initialize replay memory $D$ to capacity $N$
2. Initialize action-value function $Q$ with random weights $\theta$
3. Initialize target action-value function $\bar{Q}$ with random weights $\bar{\theta} = \theta$
4. for $\text{episode} \leftarrow 1$ to $M$ do
5. Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6. for $t \leftarrow 1$ to $T$ do
7. With probability $\epsilon$ select a random action $a_t$
8. Otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
9. Execute action $a_t$ in emulator and observe reward $r_t$ and next observation $x_t$
10. Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
11. Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
12. Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
13. Set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \bar{\theta}) & \text{otherwise} \end{cases}$
14. Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
15. Every $C$ steps reset $\bar{Q} = Q$
16. end
17. end
```

Recently, numerous modifications have been proposed to improve different aspects of the traditional Deep Q-Network. In the following, we briefly review two such exten-
The Rainbow Agent and Implicit Quantile Networks, which are relevant to our research.

Rainbow Agent

The Rainbow Agent introduced by Hessel et al. [HMVH+18] combines in a single approach several extensions that address different limitations of the traditional DQN agent. In the following, we briefly review them and discuss how they contribute to the performance of the new agent:

- **Double Q-Learning** [VHGS16]. Due to the maximization step in Equation 2.4, conventional deep Q-networks tend to select overestimated action values. Double Q-Learning tackles this issue by using the behavior network to find the best subsequent action, and the target network to estimate its quality. As a result, the new target function (i.e. $y_j$ in Algorithm 1) is computed as follows:

$$y_j = \begin{cases} r_j, & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \hat{Q}(\phi_{j+1}, \arg\max_{a'} Q(\phi_{j+1}, a'; \theta); \theta) \end{cases}$$

- **Prioritized Replay** [SQAS15]. DQN replays all transitions from the replay buffer with equal probability. However, ideally, we want to sample more frequently those observations that contribute to the learning the most [HMVH+18]. Schaul et al. [SQAS15] propose to pick transitions with a probability $p_t$ relative to the last experienced absolute temporal difference error. Furthermore, newly observed transitions are inserted into the replay buffer with the maximum priority, introducing also bias towards recent observations.

- **Dueling Networks** [WSH+15]. Traditional deep Q-network represents a single estimator for the Q-function. Wang et al. propose an alternative architecture that represents two separate estimators: one for the state value function and one for the action advantage function, which are then combined by a special aggregator. This complementary architecture leads to a better policy evaluation in the presence of numerous similar-valued actions [WSH+15].

- **Multi-step Learning** [Sut88]. In Equation 2.4, the target is defined as a sum of the intermediate reward and expected returns. In the multi-step learning, rewards over $n$ consecutive steps are truncated and added to the expected return. Multi-step targets with tuned parameter $n$ are likely to result in a faster learning [HMVH+18].

- **Distributional RL** [BDM17]. As mentioned previously, for each possible action, the deep Q-network returns a single value that represents the expected return. The original agent maximized the function, while the double deep Q-network uses the target network for approximation. Distributional reinforcement learning agents, instead of learning a single-value approximation, learn to approximate the distribution of action values. The discussed Rainbow agent, has 51 bins for each of the actions to represent the distribution of the Q-values [HMVH+18].

5The action advantage function is defined as $A^\pi(s, a) = Q^\pi(s, a) - V(s)$ and measures how much worse is the chosen action in comparison to the best possible action in the current state.
• *Noisy Nets* [FAP+17]. To explore the observation space of the environment, the DQN follows traditionally an $\epsilon$-greedy policy with a linearly decaying epsilon, which, however, has limitations for a certain types of problems (e.g. MontezumaRevenge from Atari 2600 game collection) [HMVH+18]. Fortunato et al. [FAP+17] suggest to increase the stochasticity of agent’s policy by adding parametric noise to the weights of the network. Over time, the agent can learn to ignore this noise, and exploit the current sub-optimal policy.

In the following, we continue reviewing extensions of the original Deep Q-Network agent, and present the Implicit Quantile Network agent, another example of the distributional RL approach.

**Implicit Quantile Networks**

As mentioned previously, the conventional deep Q-Network returns an expected value of the Q-function for each of the possible actions. Previous research has argued that modelling the distribution of the Q-values instead provides additional benefits [BDM17]. As a result of this observation, numerous distributional reinforcement learning agents were introduced. *C51 agent* [BDM17], an original distributional RL approach, proposed to discretize the domain of action values and compute the probability of observing rewards in each of $N$ ranges. Although, the C51 agent outperformed the original DQN agent on the ALE benchmark, the cross-entropy loss function employed by the agent can’t be generally minimized using stochastic gradient methods, which poses a concern regarding the practical application of the approach [DRBM18].

![Network architectures for DQN and recent distributional RL agents (original image by [DOSM18]).](image)

To tackle this issue, Dabney et al. introduced the *QR-DQN* agent [DRBM18] based on the concept of *quantile regression* [Koe05]. Instead of computing the distribution of action values for $N$ fixed ranges, the QR-DQN approach finds $N$ locations that have the same fixed probability. Even though this approach exceeded the performance of C51, QR-DQN still provides just a rough estimation of the distribution of return values [DOSM18].
**Implicit Quantile Network** is the state-of-the-art distributional RL agent, builds on top of QR-DQN and learns the full quantile function, which is capable of approximating any arbitrary distribution over returns [DOSM18]. This new agent halves the performance gap between the QR-DQN and the Rainbow agent, and has a potential to exceed the performance of the Rainbow approach after the integration of other orthogonal extensions like prioritized experience replay [DOSM18]. All of the discussed key differences are summarized in Figure 2.4.

**Proximal Policy Optimization**

Previously discussed reinforcement learning methods belong to the family of value-based approaches. In the following, we’ll discuss the concept behind **Proximal Policy Optimization (PPO)** agent [SWD+17], the only **policy-based** algorithm that was used in our experimental setup. Firstly, we provide essential background information on the **policy gradient methods**. Secondly, we discuss the key challenges related to the on-policy learning and how PPO tries to solve them. Finally, the concept and architecture of the agent are presented.

In contrast to the value-based methods, in policy-based approaches, the optimal policy is learned directly and is not derived from the value function [FLHI+18]. The underlying deep neural network doesn’t approximate the Q-function anymore, but represents the policy and maps states to actions. Due to such setting, these approaches have a more stable behavior [MBM+16], can be directly applied to the environments with continuous action spaces [LHP+15] and are capable of learning stochastic policies [FLHI+18].

The quality of the current policy is evaluated using the **objective function**, usually defined as [SWD+17]:

\[
L^{PG}(\theta) = E \left[ \log \pi_{\theta}(a_t | s_t) A_t \right]
\]

(2.6)

where \( \theta \) are policy parameters, \( \pi \) is a stochastic policy and \( A_t \) is the estimation of the advantage function at timestep \( t \). Building on this, the goal of the policy gradient agent is to maximize the objective function by tuning policy’s parameters. Step-by-step, the agent performs gradient ascent on these parameters and approaches the function’s optimum.

However, taking the gradient ascent step directly on such objective function may result in either very small step sizes (the learning will be slow) or very large steps (there is too much variability in the training process) [SLA+15]. The Proximal Policy Optimization agent introduces a new **clipped surrogate objective function** that improves the stability of actor’s training by limiting the size of gradient steps [SWD+17].

Firstly, instead of using the \( \log \pi \) to determine the impact of the action, PPO uses a ratio between action’s probability under the new policy and the probability of the action under the previous policy:

\[
 r_t(\theta) = \frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}
\]

(2.7)
If $r_t(\theta) > 1$, then the action is more likely now than under the old policy, and otherwise if $r_t(\theta)$ is between 0 and 1. Consequently, the new objective function is computed as:

$$L^{CPI}(\theta) = E \left[ \frac{\pi_{\theta}(a_t \mid s_t)}{\pi_{\theta,id}(a_t \mid s_t)} A_t \right] = E \left[ r_t(\theta) A_t \right]$$  \hspace{1cm} (2.8)

However, without an additional constraint, the gradient ascent step on this function can still cause the optimized policy to learn slower or diverge. One of the ways to constraint the policy update is to use KL divergence as suggested by the Trust Region Policy Optimization [SLA$^+$15]. But due to the computation inefficiency and complex implementation of this method, PPO integrates the clipping directly into the objective function:

$$L^{CLIP}(\theta) = E \left[ \min(r_t(\theta) A_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_t) \right]$$  \hspace{1cm} (2.9)

**Figure 2.5:** The practice-oriented architecture of a PPO agent.

From the architecture point of view, PPO is similar to other Actor-Critic agents [KT00]. In DRL setting this means that the agent consists of two networks: the actor network that interacts with the environment and fills the experience buffer, and the critic network that computes the value function and evaluates the actor’s choice [MBM$^+$16]. In contrast to value-based approaches, where the value function determines the policy, in PPO agent, action values are only used to compute the objective function [FLHI$^+$18]. Once the experience buffer is full, both networks are optimized using different update functions and collected experiences are discarded.

Since policy gradient methods don’t use the experience replay, they have to compensate the lacking in old transitions with a sufficient amount of new experiences. Therefore, in practice, to keep such approaches sample efficient, numerous agents running the same policy will be collecting experiences in parallel\(^6\). Once enough observations are collected, a single optimizer will update the policy and distribute it to all agents. This design is shown in Figure 2.5.

\(^6\)OpenAI Five (https://openai.com/blog/openai-five/) employs such scaled-up version of PPO in their multi-agent system.
2.2 DRL Frameworks

Recently, the emphasis of research in the field of deep reinforcement learning has shifted towards more complex agent-environment interactions [CMG+18]. As a result, having reliable and reusable reference implementations of DRL agents has become a key aspect that boosts new developments both in academic and production settings. In the following, we review a wide variety of off-the-shelf solutions that provide flexible interfaces to connect state-of-the-art DRL agents to numerous benchmark environments. We start by discussing design principles, architectures and technology stacks of three state-of-the-art frameworks evaluated in this study: a research-oriented Dopamine [CMG+18], an end-to-end production framework Horizon [GCL+18] and Ray RLib [LLM+17] that focuses on agent’s learning in distributed setting. Finally, we provide a brief overview of other promising solutions in the domain of deep reinforcement learning.

2.2.1 Dopamine

In our comparison, Dopamine framework represents the family of research-oriented DRL solutions. Different research objectives in this field demand various levels of complexity from the research software. For instance, architecture research focuses on engineering issues like scaling up distributed training and is likely to need highly-optimized modular frameworks. Contrarily, algorithmic research in the field of deep reinforcement learning benefits from simple solutions that support fast prototyping of new ideas [CMG+18]. In this landscape of DRL software, Dopamine positions itself as an intuitive framework for algorithmic research and educational purposes.

The architecture of this TensorFlow\(^7\)-based framework, shown in Figure 2.6, is mainly motivated by the following principles:

- **Self-contained and Compact**
  By keeping all the core logic of the framework in a small number of files, Dopamine

---

\(^7\)An end-to-end open source machine learning platform: https://www.tensorflow.org/.
provides an easy to comprehend platform for newcomers to the field of deep reinforcement learning. All aspects of agent’s interaction with the environment (e.g. sending actions and receiving observations) are managed by the Runner class. To ensure recovery in case of failures and future re-use of learned policies, the Checkpointer component periodically saves the experiment’s state. Additional monitoring of the experiment’s progress is done by the Logger component, which creates Tensorboard event files for the visual analysis of collected statistics. Different agent classes contain the core functionality of three commonly used value-based approaches: conventional Deep Q-Network and distributional agents, Rainbow and Implicit Quantile Networks.

- Reliable and reproducible

Ensuring consistency and reproducibility of experimental results is one of key challenges in the field of deep reinforcement learning [FLHI+18]. While variability of results due to the stochastic nature of environments and agents is inevitable, some inconsistencies can be avoided by stricter reporting standards and shareable evaluation setups [HIB+18]. By using gin-config files, Dopamine provides an easy way to share complete configurations of the agents. Several examples of such files are given in Section A. and Section B.

Testing reinforcement learning systems is a new area with no established best practices [GCL+18]. Dopamine contributes to this problem by providing a complete suite of test cases with a code coverage of over 98% [CMG+18].

Despite all of these benefits, such flexible and comprehensive architecture comes at a cost of a rather limited functionality. In contrast to other DRL solutions presented in this work, Dopamine is tailored to interact only with simulated environments, can’t operate in distributed and offline settings and provides a small number of agents.

2.2.2 Horizon

In contrast to the previously discussed Dopamine framework, Horizon positions itself as the first end-to-end platform to solve production reinforcement learning problems, considering that this approach is likely to transform how autonomous systems are built. Horizon tackles datasets containing millions of observations, and cases where simulators can’t be used to evaluate the policy, and learning has to be done with care to preserve the established production system and user’s experience [GCL+18]. Currently known cases of applying policies learned with Horizon in production include [GCL+18]:

- Sending push notifications. Facebook sends notifications to people informing them about the most recent updates about their friends, posts, events etc. To make sure that only the most relevant notifications are being sent, Facebook filters notification candidates using Deep Q-Network trained with Horizon. The agent maps a set of features about the person and the notification to Q-values for two actions: send and drop, and attempts to send only those notifications that result in user’s interaction and activity on Facebook.

8Dopamine’s implementation of the Rainbow agent includes only multi-step learning, prioritized experience replay and distributional learning.

Adaptive bitrate for 360-degree videos. One of the key properties of reinforcement learning agents is their ability to quickly adapt to the newly observed data and make accurate predictions on-the-fly. This advantage has allowed to improve the image quality of 360-degree videos on Facebook. The trained agent observes the amount of available bandwidth and the amount of video buffered, and decides whether the quality of the playback can be increased or not.

In the following, we present Horizon’s key components and discuss other production challenges addressed by this framework. Figure 2.7 provides a complete overview of Horizon’s workflow and the place of the discussed components in it.

Figure 2.7: Horizon’s DRL pipeline for production environments$^{10}$.

**Data Preprocessing**

As mentioned previously, conventional deep reinforcement learning agents are trained on MDP transitions that contain the current state, the chosen action, the consecutive state and the reward. Since traditional benchmarks like OpenAI Gym use simulators to provide the feedback, such transitions can be created immediately and stored in the experience buffer for learning. However, in production setting, where the feedback loop is slow and all observations are logged immediately, it may take up to several hours or even days to get the reward for the chosen action. Therefore, Horizon provides an Apache Spark data preprocessing pipeline (called the Timeline pipeline) that analyzes production logs and creates transitions used by reinforcement learning agents.

$^{10}$Adapted from https://engineering.fb.com/ml-applications/horizon/.
Feature Normalization

Production data is frequently sparse, noisy and arbitrarily distributed. Since it has been shown that the neural network learns better when trained on normally distributed features [IS15], a feature normalization workflow is crucial for an end-to-end DRL system. As a result, Horizon analyzes the training data and for each of the features, derives a normalization function. Instead of regenerating the entire dataset, Horizon integrates this normalization function into the PyTorch neural network that will normalize each sample of the data during the forward pass.

Data Understanding Tool

Applying deep reinforcement learning methods to ill-formulated environments may regress evaluation metrics and push development process into tuning of irrelevant factors. However, finding a proper formulation of production sequential decision making tasks may be problematic for inexperienced developers. Consequently, the development community could benefit from a tool that can accelerate the problem formulation. Horizon suggests to learn a model of the environment from collected production logs, and use a combination of the model and heuristics to check the problem for validity and identify important features for the construction of the environment.

Implemented DRL Agents

To support environments with both (very large) discrete and continuous action spaces, Horizon provides implementation of several value-based and policy-based agents. For discrete action domains with a manageable number of actions, Horizon provides a traditional Deep Q-Network and a partial Rainbow agent\textsuperscript{11}. To handle very large discrete action domains that potentially contain ephemeral actions, Horizon introduces Parametric-Action DQN, a variant of DQN agent that can filter out invalid actions based on a set of features. For continuous action spaces, Horizon implements Deep Deterministic Policy Gradients [LHP+15] and Soft Actor Critic [HZAL18] agents, which are both simple and effective.

Training in Production Setting

Once the input data has been preprocessed and normalized, Horizon utilizes PyTorch\textsuperscript{12} multi-GPU functionality to efficiently perform the distributed training on extremely large samples of data. In contrast to simulated environments, training the RL agent online right from the start can be undesirable. The initial random policy deployed to the production system may expose users to an untested experimental policy [GCL+18] or in case of safety critical systems, have irreversible effects on the system [LMK+17]. Therefore, Horizon suggests to learn the initial policy offline on the data collected by the current non-RL production policy. Once the trained policy achieves a certain level of rewards,

\textsuperscript{11}Horizon’s implementation of the Rainbow agent includes only double Q-learning, dueling architecture and multi-step learning.

\textsuperscript{12}An open source machine learning framework: https://pytorch.org/.
it will be deployed on parts of the production data and will be trained in a full online setting.

**Counterfactual Policy Evaluation**

Similarly to the previously made point, the policy being trained can’t be evaluated directly on the production environment due to its unpredictable behavior. Furthermore, designing an accurate simulator for real-world environments is a problematic task due to the complexity of the problem and usually large number of integrated components [MNN+19]. As a consequence, Horizon uses *counterfactual policy evaluation (CPE)* methods which are capable of estimating the quality of the policy based on historical data without deploying it to the production system [WAD17]. During the training process, Horizon executes one of the following CPE estimators: step-wise direct method estimator, step-wise importance sampling estimator [HT52], step-wise doubly-robust estimator [DLL11], sequential doubly-robust estimator [JL15], sequential weighted doubly-robust estimator [TB16] or MAGIC estimator [TB16]. The general idea behind these techniques is to learn the estimate of environment’s reward function to predict rewards that are not observed but are likely to occur [GCL+18]. For more details about each of the approaches, we refer the interested reader to the cited works.

**Optimized Model Serving**

In real-world applications, where the production environment can be run in a cluster containing thousands of machines [GCL+18], it is important to ensure an efficient and robust serving of the model. Horizon approaches this challenge by using PyTorch and the ONNX [Exc18] format to create a portable package containing the agent’s configuration. The policy deployed to numerous machines will gather new experiences into the production log, which can be fed to the Timeline preprocessing pipeline and be used in next iterations of the learning process. This closed interaction loop is depicted in Figure 2.8.

![Figure 2.8: A high-level representation of Horizon’s production feedback loop](https://engineering.fb.com/ml-applications/horizon/)

---

13Adapted from https://engineering.fb.com/ml-applications/horizon/.
2.2.3 Ray RLlib

As mentioned previously, the deep reinforcement learning workflow generally consists of two major phases: collection of experiences and policy optimization, where each of the steps offers opportunities for distributed execution. In theory, one can achieve parallelism by applying a specific distributed system to each step and stitching them together. However, in practice, this approach is untenable due to the tight coupling of the components and high communication costs between systems [MNW+18]. Consequently, current DRL libraries tend to implement DRL agents as single strongly connected processes and apply parallel methods to the complete procedure instead of individual modules. Unfortunately, this strategy makes such implementations difficult to extend, combine and reuse [LLM+17]. Previous research has shown that despite seemingly large differences in design of DRL agents, several components like policy evaluation and gradient based optimization appear in most of them (refer to Figure 2.9 for more details). Therefore, there is an obvious need for a solution that implements efficient abstract composable parallel components.

<table>
<thead>
<tr>
<th>Algorithm Family</th>
<th>Policy Evaluation</th>
<th>Replay Buffer</th>
<th>Gradient-Based Optimizer</th>
<th>Other Distributed Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQNs</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Policy Gradient</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Off-policy PG</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Model-Based Planning</td>
</tr>
<tr>
<td>Model-Based/Hybrid</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Derivative-Free Optimization</td>
</tr>
<tr>
<td>Multi-Agent</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>MCTS, Derivative-Free Optimiz</td>
</tr>
<tr>
<td>Evolutionary Methods</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AlphaGo</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.9: Overview of common components in popular deep reinforcement learning agents (original image by [LLM+17]).

One of the key issue that this framework has to solve is to adequately handle highly irregular computation patterns at different stages of DRL agent’s workflow [LLM+17]. Firstly, the duration of algorithm’s tasks can range from milliseconds (e.g. taking an action) to hours or even days (e.g. receiving feedback from the environment). Secondly, different phases of training require heterogeneous hardware (e.g. sampling of experiences often runs on multiple CPUs, while policy optimization on GPU). Finally, RL algorithms include both stateless tasks for fine-grained simulation and data processing, and stateful computations to maintain replay buffers and network parameters. To fulfill these requirements, RLlib structures distributed DRL components around the principle of logically centralized and hierarchical control model built on top of Ray [MNW+18], a framework for distributed execution of Python tasks. In the following, we revisit the Ray implementation of the hierarchical control model and discuss how popular DRL agents can be built using RLlib’s abstract parallel components.

Ray’s Hierarchical Parallel Task Model

In contrast to the traditional distributed model used to achieve parallelism in current DRL libraries, RLlib suggests to use the hierarchical version of the logically centralized control model. As depicted in Figure 2.10, the essence of this approach is to encapsulate the entire distributed control logic in a single driver program that delegates algorithm’s
Figure 2.10: An overview of state-of-the-art distributed control models (adapted from [LLM+17]). Red dotted edges denote the control flow, while dark blue solid edges indicate data flow.

sub-tasks to other processes executed in parallel. These processes, called workers, may hold the agent’s state (e.g. policy network, replay buffer or simulator’s state), but don’t run any computations until the driver tells them to. In the hierarchical version of this model, individual workers are allowed to operate as drivers and delegate their tasks to further sub-workers. Such hierarchical delegation of control supports a straightforward nesting of distributed components within each other, which is crucial for tightly coupled modules of DRL agents. In addition to the simple and flexible architecture, logically centralized control models provide high performance due to the majority of data transfer happening directly between workers, not passing through the driver, which is the central bottleneck of the system.

Figure 2.11: Architecture of the task-based programming model used in Ray (original image by [MNW+18]).

Although within a single machine the presented model can be implemented with thread-pools and shared memory, RLlib framework aims at scaling to larger clusters of nodes with heterogeneous hardware [LLM+17]. Ray’s distributed scheduler provides a natural fit for implementing the logically centralized hierarchical control in such setting. Each of the nodes in the cluster is provided with a local scheduler, that tries to assign tasks locally, unless the current node is overloaded (e.g. the task queue exceeds a predefined threshold) or node’s resources don’t satisfy task’s requirements (e.g. task requires GPU support) [MNW+18]. If the task can’t be scheduled on the current node, the local scheduler
forwards it to the centralized global scheduler that analyzes system’s load and task’s requirements to find a suitable worker. Ray supports both stateless and stateful tasks (called actors), which can launch more remote tasks and actors, satisfying the need for hierarchical delegation of control. To declare a remote task or class and specify resources for it, one simply has to add a Ray decorator. Within a single node, all tasks can access the shared memory store which supports zero-copy data sharing between tasks. To access data of the task running on a remote node, the current worker, firstly, has to look up the location of the data in Global Control Store, which is a key-value store with the control information of the entire system. And, secondly, once the data is accessible, it is transmitted directly between nodes using multiple TCP connections. We put all these concepts together in Figure 2.11, which shows Ray’s components and connections between them.

Abstract Parallel DRL Components

The implementation of a DRL agent in RLlib starts with the definition of the underlying policy graph, which provides the definition of the neural network, as well as the interface to access network’s parameters and compute the loss function [LLM+17]. This policy graph can be specified in any deep learning framework, while RLlib currently supports TensorFlow and PyTorch. For the interaction with the environment, RLlib provides a PolicyEvaluator class that wraps defined policy graph and environment, and provides the sample method to collect experiences [LLN+17]. This class, common for all agents, can be instantiated as a Ray remote actor and replicated across multiple workers for parallel collection of experiences. Currently, RLlib supports interaction with OpenAI Gym [BCP+16], user-defined environments, and batched simulators such as ELF [TGS+17].

The final component of RLlib agents is the algorithm-independent policy optimizer, responsible for the performance-critical tasks of experience sampling, network’s parameter updates and management of replay buffers. To distribute the collection of experiences, the optimizer launches numerous remote evaluators discussed previously. Then, depending on the available resources, the optimizer either computes the gradients locally or delegates this task to remote evaluators. Once the gradients are obtained, it updates the parameters of the local copy of the policy graph and broadcasts them to all remote evaluators running this policy.

Using these abstractions, RLlib provides reference implementations of numerous state-of-the-art value-based (e.g. DQN [MKS+13], Rainbow [HMVH+18]), policy-based (e.g. A2C/A3C [MBM+16], PPO [SWD+17]) and hybrid (e.g. DDPG [LHP+15], SAC [HZAL18]) agents, as well as proposes several highly distributed variations of established approaches (e.g. Ape-X DQN [HQB+18], IMPALA [ESM+18], APPO [SWD+17]). This design separates distributed computations from agent-specific policy and loss definitions, which allows algorithm developers to tune their agent on different isolated levels. They can either focus on the core algorithmic properties of the agent and adjust the policy graph, or experiment with distributed sampling patterns and swap optimizers based on available hardware or algorithm’s features [LLM+17].
2.2.4 Landscape of DRL Frameworks

By no means exhaustive, this overview highlights the heterogeneity of solutions that provide DRL functionality and the value of evaluations such as ours.

1. Amazon SageMaker RL\(^{14}\) provides a modular platform for distributed DRL that integrates state-of-the-art deep learning frameworks (e.g. TensorFlow, Apache MXNet) and RL toolkits (e.g. Intel Coach, Ray RLlib), and connects them to traditional simulated environments (e.g. OpenAI Gym).

2. DeeR [FL+16] is a modular research-oriented framework that supports easy modifications of agent’s components.

3. Dopamine [CMG+18] provides a compact, stable and flexible framework for fundamental algorithmic research in the field of deep reinforcement learning.

4. ELF [TGS+17] is an extensive, lightweight and flexible platform for fundamental RL research in the field of real-time strategy games.

5. Garage is a framework for algorithmic research built on top of RLLab [DCH+16], supporting both TensorFlow and PyTorch neural networks.

6. Horizon [GCL+18] is an end-to-end reinforcement learning platform for large-scale production environments.

7. Huskarl [Sal19], a recently introduced modular DRL framework, integrates the popular deep learning library Keras to provide an intuitive API for fast prototyping of new approaches.

8. Intel Coach [CLNE17] focuses on increasing the speed of learning by running multiple instances of (asynchronous) agents in parallel on many CPU cores.


11. PARL\(^{15}\) focuses on the distributed training of DRL agents and easy implementation of new environments and agents from provided abstract components.

12. PyBrain [SBW+10] is a modular python library with limited DRL functionality, aimed at the newcomers to the field of reinforcement learning.

13. Ray RLlib [LLM+17] is a DRL solution built on top of Ray [MNW+18] optimized for distributed learning. The toolkit provides a large number of agents composed from a set of abstract modules.


\(^{15}\)https://github.com/PaddlePaddle/PARL
15. **rlpyt** [SA19] is a novel framework that implements state-of-the-art agents from three families of DRL approaches: deep Q-learning, policy gradients and Q-value policy gradients, on top of a shared and optimized infrastructure.

16. **TensorForce** [KSF17] is a TensorFlow-based framework with an emphasis on modular design and straightforward application in research and practice.

## 2.3 Benchmark Environments

Recent developments in the domain of deep reinforcement learning have encouraged the introduction of numerous toolkits for the comparison of DRL concepts. The list of available benchmarks includes, but is not limited to, Arcade Learning Environment (ALE) [BNVB13], RLLab [DCH16], RLPy [GDK+15], RL-Glue [TW09], PyBrain [SBW+10], Malmo [JHHB16], Iroko [RPB18] etc. In the following, we revisit two benchmarks used in our experimental setup: OpenAI Gym [BCP+16] and Park [MNN+19], and briefly stop on the AI Safety Gridworlds benchmark [LMK+17] which highlights the evaluation of safety concerns, which is an essential aspect for production-oriented DRL solutions.

### 2.3.1 OpenAI Gym

OpenAI Gym benchmark provides a flexible interface to access a wide range of tasks (called *environments*) introduced by previous toolkits. In addition to these, OpenAI Gym launched a website\(^\text{16}\) where researchers in the field of reinforcement learning can share their evaluation results and the configuration of evaluated agents. The collection of benchmark tasks together with a platform for sharing results encourages peer review and a better reproducibility of experimental results.

The toolkit provides a clear interface for the environment, but no strict specification of the agent, which allows users to implement different types of learning procedures (e.g. *online* and *batch update* settings). The environment interface is optimized for episodic setting, which means that agent’s interaction with the task is divided into *episodes* of fixed length. At the start, the agent observes a randomly sampled initial state, and proceeds until it reaches the terminal state (e.g. *game over*). The goal of the agent in this setting is to maximize the total reward for the episode and discover the optimal policy in the smallest number of episodes.

The following minimal code snippet shows the implementation of a single episode of OpenAI Gym’s environment.

```python
env = gym.make(env_name)
obs = env.reset()
done = False
while not done do
    action = agent.get_action(obs)
    obs, reward, done, info = env.step(action)
```

\(^{16}\)https://gym.openai.com/
Firstly, we initialize the environment by calling the `make` function. Secondly, the initial state of the environment is returned by `env.reset()`. Then, an instance of the agent class, implemented by the user, computes an action to be taken in the current state. Finally, the environment takes this action as an input, transitions into the next state and returns relevant information to the agent.

All in all, OpenAI Gym provides over 100 environments implementing the described interface. The history of all changes to the environments is strictly recorded to avoid inconsistencies when comparing evaluation results. Environments vary in the level of difficulty and include:

- **classic control and toy text tasks**: small traditional tasks used in RL research (e.g. CartPole, Acrobot);
- **algorithmic tasks**: simple computation tasks demanding the proper memory management from the agent (e.g. copy symbols, reverse sequences);
- **Atari games**: a collection of Atari 2600 arcade games provided by the ALE benchmark (e.g. SpaceInvaders, MontezumaRevenge);
- **robotic problems**: control a robot in a simulator (e.g. hand manipulation tasks).

### 2.3.2 Park Project

A wide range of sequential decision making problems can be found in real-world applications dealing with networking, database management, and distributed computing [MNN+19]. These tasks introduce new challenges for reinforcement learning approaches that are not typical for well-studied simulated environments. Such challenges include extremely large and time-varying state and action spaces, new types of structured observation spaces (e.g. graphs representing network topologies), and highly-distributed environments running on multiple nodes in a cluster [GCL+18]. Despite a great potential for applying reinforcement learning in a production setting, there is still relatively little research done in this domain [MNN+19]. By providing a flexible platform for fast prototyping and a suite of representative real-world environments, Park project encourages future research on algorithms and application of RL in real-world applications.

![Figure 2.12: Park's request-response architecture (adapted from [MNN+19]).](image)

To ensure a straightforward connection to existing environments and seamless integration of new systems, Park follows a traditional request-response design pattern. Each backend system (environment) defines events that trigger an MDP transition. As soon as such
event occurs, the system sends an RPC request with the current state of the environment and rewards for the last step to the Park server. Upon receiving the request, the server forwards the information about the environment to the agent. Once the agent picks the next action, Park server sends a response to the back-end system, and the interaction loop continues. Similarly to the OpenAI Gym, the implementation of the agent can be arbitrary, and can be connected to the Park server via a common interface (\texttt{make - reset - step}). Such design, also shown in Figure 2.12, hides the complexity of underlying system from the implementation of RL agent, which makes all these components easily replaceable.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Type</th>
<th>State space</th>
<th>Action space</th>
<th>Reward</th>
<th>Step time</th>
<th>Challenges (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive video streaming</td>
<td>Real/sim</td>
<td>Past network throughput measurements, playback buffer size, portion of unwatched video</td>
<td>Bitrate of the next video chunk</td>
<td>Combination of resolution and stall time</td>
<td>Real: ~3s Sim: ~1ms</td>
<td>Input-driven variance, slow interaction time</td>
</tr>
<tr>
<td>Spark cluster job scheduling</td>
<td>Real/sim</td>
<td>Cluster and job information as features attached to each node of the job DAGs</td>
<td>Node to schedule next</td>
<td>Runtime penalty of each job</td>
<td>Real: ~5s Sim: ~5ms</td>
<td>Input-driven variance, state representation, infinite horizon, reality gap</td>
</tr>
<tr>
<td>SQL database query optimization</td>
<td>Real</td>
<td>Query graph with predicate and table features on nodes, join attributes on edges</td>
<td>Edge to join next</td>
<td>Cost model or actual query time</td>
<td>~5s</td>
<td>State representation, reality gap</td>
</tr>
<tr>
<td>Network congestion control</td>
<td>Real</td>
<td>Throughput, delay and packet loss</td>
<td>Congestion window and pacing rate</td>
<td>Combination of throughput and delay</td>
<td>~10ms</td>
<td>Sparse space for exploration, safe exploration, infinite horizon</td>
</tr>
<tr>
<td>Network active queue management</td>
<td>Real</td>
<td>Past queuing delay, enqueue/dequeue rate</td>
<td>Drop rate</td>
<td>Combination of throughput and delay</td>
<td>~50ms</td>
<td>Infinite horizon, reality gap</td>
</tr>
<tr>
<td>TensorFlow device placement</td>
<td>Real/sim</td>
<td>Current device placement and runtime costs as features attached to each node of the job DAGs</td>
<td>Updated placement of the current node</td>
<td>Penalty of runtime and invalid placement</td>
<td>Real: ~2s Sim: ~10ms</td>
<td>State representation, reality gap</td>
</tr>
<tr>
<td>Circuit design</td>
<td>Sim</td>
<td>Circuit graph with component ID, static parameters as features on the node</td>
<td>Transistor sizes, capacitance and resistance of each node</td>
<td>Combination of bandwidth, power and gain</td>
<td>~2s</td>
<td>State representation, sparse space for exploration</td>
</tr>
<tr>
<td>CDN memory caching</td>
<td>Sim</td>
<td>Object size, time since last hit, cache occupancy</td>
<td>Admin/drop</td>
<td>Byte hits</td>
<td>~2ms</td>
<td>Input-driven variance, infinite horizon, safe exploration</td>
</tr>
<tr>
<td>Multi-dim database indexing</td>
<td>Real</td>
<td>Query workload, stored data points</td>
<td>Layout for data organization</td>
<td>Query throughput</td>
<td>~30s</td>
<td>State/action representation, infinite horizon</td>
</tr>
<tr>
<td>Account region assignment</td>
<td>Sim</td>
<td>Account language, region of request, set of linked websites</td>
<td>Account region assignment</td>
<td>Scheduling cost in the future</td>
<td>~1ms</td>
<td>State/action representation</td>
</tr>
<tr>
<td>Server load balancing</td>
<td>Sim</td>
<td>Current load of the servers and the size of incoming job</td>
<td>Server ID to assign the job</td>
<td>Runtime penalty of each job</td>
<td>~1ms</td>
<td>Input-driven variance, infinite horizon, safe exploration</td>
</tr>
<tr>
<td>Switch scheduling</td>
<td>Sim</td>
<td>Queue occupancy for input-output port pairs</td>
<td>Bijection mapping from input ports to output ports</td>
<td>Penalty of remaining packets in the queue</td>
<td>~1ms</td>
<td>Action representation</td>
</tr>
</tbody>
</table>

\textbf{Figure 2.13:} Overview of computer system environments supported by Park platform (original image by [MNN+19]).

In total, Park supports 12 computer system environments: seven of the environments rely on real systems in the back-end (e.g. network congestion control environment uses CCP platform [NCR+18] in the background) and for remaining five, high-quality simulators are provided. By relying on real-world data traces, simulators are able to imitate production environments as precisely as possible (e.g. CDN memory caching environment uses an open dataset containing 500 million requests [BSHB17] to simulate the behavior of the cache). A complete overview of supported environments is given in Figure 2.13.
2.3.3 AI Safety Gridworlds

An increasing number of research dealing with applying DRL approaches in production environments clearly indicates that more and more AI systems will be deployed in real-world applications. As a result of this trend, numerous works regarding the safety of AI approaches have been released \[\text{[JJD}^+16]\][\text{SSSS16}][\text{BTSK17}]. Although in this research, we don’t evaluate safety aspects of the selected frameworks, discovering safety flaws in the implementation of DRL agents will be a crucial part of future improvements. Therefore, in the following we briefly present the AI Safety Gridworld benchmark \[\text{[LMK}^+17]\] that provides a suite of basic environments illustrating different safety concerns.

In particular, the benchmark focuses on robustness and specification problems, considering challenges such as safe interruptibility \[\text{[OA16]}\], avoiding side effects \[\text{[AOS}^+16]\], absent supervisor \[\text{[Arm17]}\], reward gaming \[\text{[CA]}\], self-modification \[\text{[LMK}^+17]\], distributional shift \[\text{[QCSSL09]}\], robustness to adversaries \[\text{[ACBFS02]}\] and safe exploration \[\text{[PS14]}\]. In order to clarify the core of each problem, the benchmark provides several simple environments called \textit{gridworlds}. Each environment is represented by a grid of cells, the agent always occupies one tile and can either interact with another object in the cell or move to an adjacent position. In addition to such grid, each environment has two functions: a standard \textit{reward function} exposed to the agent, and a (safety) \textit{performance function} hidden from the agent which captures the safety aspects of the problem. For a given problem, an agent is considered to be "safe", if by maximizing the known reward function, it also reaches the optimum for the performance function.

![Figure 2.14: The absent supervisor environment (original image by \[\text{[LMK}^+17]\)].](image)

To keep it short, we present one example of an environment related to the issue of \textit{absent supervisor}. The core idea of this concern is that a capable agent can learn to "fake" better performance during supervised tests, and show poor results when put into production. To help uncover this flaw, a gridworld shown in Figure 2.14 was proposed. The agent starts at cell A and gets a reward for reaching the goal G. When stepping on tile P, the agent receives a punishment only if the supervisor S is present. In this setting, a safe agent has to always avoid the punishment by taking the longer route regardless of supervisor’s absence.

While such environment is highly abstract and simple, it captures the essence of a problem and can be used as a minimal sanity check: if a general RL model fails to behave properly in such gridworld, then we can’t expect it to be safe in more complex real-world environments.
2.4 Summary

In this chapter, we provided theoretical prerequisites essential for the understanding of our comparison results, which will be presented in the following sections. To start with, the concept behind reinforcement learning (RL), a field of machine learning that deals with sequential decision making tasks, was revisited. We introduced major components of an RL system such as agent, environment, policy, reward, value functions and model, and discussed how RL tasks can be formalized using finite Markov Decision Processes (MDPs). Later, several learning settings and strategies were presented that allow us easily differentiate between various types of RL algorithms. To conclude the section, we provided a detailed description of RL algorithms known as deep reinforcement learning agents, which use deep neural networks to find the optimal policy. In particular, we focused on several value-based agents (e.g. Deep Q-Network, Rainbow agent and Implicit Quantile Networks) and policy-based agents (e.g. Proximal Policy Optimization), which are used in our experimental setup.

<table>
<thead>
<tr>
<th>Property</th>
<th>Dopamine</th>
<th>Horizon</th>
<th>Ray RLlib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework’s focus</td>
<td>Research Education</td>
<td>Production</td>
<td>Distributed learning</td>
</tr>
<tr>
<td>Environments</td>
<td>OpenAI Gym User-defined</td>
<td>OpenAI Gym User-defined</td>
<td>OpenAI Gym ELF User-defined</td>
</tr>
<tr>
<td>Agents</td>
<td>Value-based</td>
<td>Value-based</td>
<td>Value-based</td>
</tr>
<tr>
<td>Setting</td>
<td>Online</td>
<td>Online</td>
<td>Online</td>
</tr>
<tr>
<td></td>
<td>Offline</td>
<td></td>
<td>Offline</td>
</tr>
<tr>
<td>Supported DL libraries</td>
<td>TensorFlow</td>
<td>PyTorch</td>
<td>PyTorch TensorFlow</td>
</tr>
<tr>
<td>GPU support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed learning</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Code base (LoC)</td>
<td>35 files (4798)</td>
<td>126 files (23,434)</td>
<td>2k files (590,924)</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of framework’s properties for Dopamine, Horizon and Ray RLlib.

In Section 2.2, we argued that the recent shift in research objectives towards more complex agent-environment interactions demands an introduction of reliable and reusable implementations of DRL agents. To increase community’s awareness of existing off-the-shelf solutions and their properties, we provided an overview of a broad range of DRL frameworks. Three frameworks used in our evaluation: Dopamine, Horizon and Ray RLlib, were discussed in more details, whereas the main focus was put on the architecture, implemented functionality and technology stack of the solution. Table 2.1 provides a summary of our analysis.

The conclusion of the chapter was dedicated to the benchmark environments used to evaluate deep reinforcement learning concepts. Especially, we discussed a traditional OpenAI Gym benchmark, containing over 100 gaming, classic control and algorithmic environments; newly introduced Park benchmark, consisting of 12 representation production
environments; and the *AI Safety Gridworlds* benchmark, which wasn’t used in our evaluation, but introduces approaches towards testing safety of DRL, which we expect will be relevant in future evaluations of DRL solutions.
3 Design of the Evaluation Platform

In the following chapter, we elaborate on the goals set in Chapter 1 and define precise research questions addressed in this evaluation. Furthermore, in Section 3.2, we present one of our main contributions, a platform\(^1\) that should support a more effortless evaluation of deep reinforcement learning frameworks.

3.1 Research Questions

This research is the first attempt to compare several off-the-shelf deep reinforcement learning solutions. We acknowledge the fact, that integration of such frameworks into real-world applications requires a thorough evaluation of source code’s quality and maintainability, framework’s performance and memory footprint [BRSS15], robustness and safety of implemented algorithms [LMK+17]. However, due to time and resource limitations of this research, we mainly focus on the performance and hardware utilization aspects of selected solutions. In particular, this work addresses the following research questions:

1. Which of the selected frameworks has the lowest training time and achieves the quickest convergence for a given problem and selected common model?
2. How well can these off-the-shelf solutions utilize different (multiple) processing units?
3. How does the number of agent’s checkpoints saved to the primary storage during the training process affect the framework’s performance?
4. Which of the DRL algorithms implemented by the framework has the best performance for a given problem?
5. Once the model has learned, how fast can the framework perform inference (i.e. interact with the environment to solve the given task)?

3.2 Design of the Evaluation Platform

The system design, depicted in Figure 3.1, is guided by the following two principles:

\(^1\)https://github.com/pshevche/drl-frameworks
• **Extensibility.** A growing landscape of software solutions in the field of deep reinforcement learning makes it nearly impossible to conduct an exhaustive evaluation. Therefore, in our implementation we focus on providing a comprehensible infrastructure as a base for future studies in the field of deep reinforcement learning frameworks. Especially, the proposed design is tailored for seamless integration of new frameworks and environments, as well as for further extension of selected solutions. We envision that having a design able to integrate multiple frameworks can also assist in tasks beyond those we study, such as the analysis of statistical soundness of models from different platforms.

• **Portability.** Ensuring reproducibility of evaluation results in the domain of deep reinforcement learning is a challenging task due to non-deterministic behavior of environments and models [FLHI+18]. Although the proposed platform doesn’t tackle problems caused by randomness in frameworks’ components, its design follows some of the best practices to ensure consistency in other parts of the evaluation setup like experiment’s workflow and dependency management.

![Image of the platform design](image)

**Figure 3.1:** Design of our platform for evaluation of deep reinforcement learning frameworks.

The entry point for evaluating with the presented platform is the `runner` script. The module’s sole responsibility is running a single experiment in the corresponding framework and reporting obtained statistics. The evaluation platform makes use of frameworks’ configuration files, which allow us to reuse tuned parameters and keep experiment’s workflow consistent. Even though these configuration modules offer a powerful tool to manipulate most of the framework’s settings, it is sometimes required to re-implement some of the framework’s components. Extending framework’s functionality rather than modifying the existing source code is a central virtue of our evaluation platform. We believe that keeping legacy code intact whenever possible ensures a better understanding of the experiment’s behavior and supports a more effortless migration to the newer versions of the evaluated frameworks when needed. During the implementation of described evaluation platform, the extensions summarized in Figure 3.2 were made to the frameworks’ standard components.

Furthermore, to simplify the system’s setup process, we utilize the advantages provided
Figure 3.2: Overview of extensions done in our work to the frameworks’ standard components.

by Anaconda\(^2\) environment management system and Docker\(^3\) container platform. As a result of using Anaconda’s configuration files, we can ensure a consistent dependencies base for the evaluated frameworks, as well as a simple integration of new frameworks. Finally, running our system inside the Docker container makes this evaluation platform easily shareable among developers and researchers, and provides a foundation for future integration with real-world applications.

\(^2\)https://docs.conda.io/projects/conda/en/latest/
\(^3\)https://www.docker.com/
4 Experimental Setup

In the following chapter we provide all the information required to reproduce the evaluation results reported in this research:

- **Section 4.1** is dedicated to the description of two environments used in the experiments: CartPole environment from OpenAI Gym benchmark [BCP+16], and Query Optimizer environment from Park benchmark [MNN+19]. Furthermore, we briefly revisit the *Join Order Benchmark* [LGM+15], an integral part of the Park’s Query Optimizer learning task.

- In **Section 4.2**, the architecture of underlying neural networks is presented, as well as some extra details regarding the selected hyper parameters.

- Finally, we conclude the chapter by providing a detailed specification hardware and software resources used, in **Section 4.3**.

4.1 Benchmarks

In our work, we evaluate the performance of DRL algorithms implemented by selected frameworks against two benchmark tasks: CartPole and Query Optimizer. The former represents classic control tasks provided by the OpenAI Gym benchmark traditionally used to evaluate deep reinforcement learning concepts. The latter environment is an example of a real-world sequential decision making task from computer systems, supported by the production-oriented Park benchmark. In the following, we explain such essential components of these environments like the structure of state and action spaces, reward functions, and in case of Query Optimizer environment, provide a brief description of underlying relations and queries.

4.1.1 OpenAI Gym’s CartPole Environment

The OpenAI Gym’s implementation of the CartPole environment closely follows the definition of this classic control problem proposed by Barto et al. [BSA83]. Figure 4.1 shows a schematic representation of this task. A cart is free to move within the bounds of a frictionless track. The lower end of the pole is attached to the cart in a way that allows it to move only in the vertical plane of the cart and the track. The goal of the agent is to prevent the pole from falling by applying impulsive “left” or “right” forces to the cart at discrete timesteps. The state of the cart-pole system at any given point can be represented by four values:

- \( x \) – position of the cart on the track;
• $v$ – velocity of the cart;
• $\theta$ – the pole angle;
• $\omega$ – angular velocity of the pole.

In the OpenAI Gym’s version of the problem, for each timestep that the pole remains upright, an agent receives a reward of $+1$. The episode of the environment ends either when the maximum steps per episode are reached, or when the position of the cart $x$ is more than 2.4 units from the center or the pole is more than 15 degrees from the vertical axis.

Figure 4.1: A schematic representation of cart-pole control task (adapted from [BSA83]).

The CartPole task presented in this subsection is considered to be a rather simple control problem. Previous research has shown that even a random search in weight space can quickly discover satisfactory coefficients for a neural controller that can successfully manipulate the cart [GS93]. Therefore, finding an optimal policy for the agent is not sufficient to reason about effectiveness of the framework’s implementation of the approach. However, because of its simplicity, CartPole is a perfect environment for fast evaluation of different aspects of DRL solutions.

4.1.2 Park’s Query Optimizer Environment

For evaluation of deep reinforcement learning frameworks, we selected query optimization as a representative sequential decision making task provided by the Park benchmark. Traditional query optimizers strive to improve the query execution time by re-ordering query operators based on a combination of complex heuristics. Since these heuristics remain constant over time, optimizers are not able to adapt and improve their performance, which makes query optimization a perfect task for optimization through reinforcement learning [KYG+18]. As a result of the observation that join order (apart from algorithm selection) has a major effect on the query execution time, Park platform has reduced the query optimization problem to finding an optimal ordering of the joined relations.
At each step of the environment, the agent observes a graph representing the current state of the query. Nodes of the graph correspond to the tables that remain to be joined, and edges connect nodes according to given join filters. Based on this observation, the agent chooses an edge in the query graph, which corresponds to a pair of tables to be joined next. Park supports several reward options for the query optimization environment such as cost estimation after each chosen action or the final query execution time for the entire plan [MNN+19]. In our research, the former option is used. The underlying cost model is based on the non-linear $CM_2$ model proposed by [KYG+18]. The model estimates the number of memory accesses required by a hash or nested loop join with a main memory limit $M$. The function adds extra costs when the data has to be partitioned or when the query execution falls back to a nested loop join:

$$c_{join} = \begin{cases} 
    c(O_l) + c(O_r) + |O|, & \text{if } |O_l| + |O_r| \leq M \\
    2 \cdot (c(O_l) + c(O_r)) + |O|, & \text{if } \min(|O_l|, |O_r|) \leq M \\
    c(O_r) + \left\lceil \frac{|O_r|}{M} \right\rceil \cdot c(O_l) + |O| & 
\end{cases}$$  \hspace{1cm} (4.1)

The described cost estimation is computed by the Calcite [BCRH+18] query optimization framework which is connected to an instance of the Postgres\textsuperscript{1} database. More information on the database schema and benchmark queries will be provided in the following subsection.

![Figure 4.2: Schematic representation of a model with action pruning.](image)

As mentioned previously, one of the Park’s goals is to encourage algorithmic research for environments with uncommon observation types like graph structures in the query optimization task. This, however, poses an issue when using selected deep reinforcement learning frameworks with the Park benchmark. Since OpenAIGym has emerged as a standard evaluation benchmark, commonly used DRL solutions are only tailored to support OpenAIGym’s observation types. Therefore, in order to make evaluation against the

\textsuperscript{1}https://www.postgresql.org/
Park benchmark possible, several extensions were made both to the original environment and frameworks’ components. Firstly, significant efforts were made to find a valid mapping between the graph structure of the query optimization environment and observation shapes supported by all of the evaluated frameworks. Luckily, graph structures can be described in numerous ways, and in this study we represented the query graph as a linearized adjacency matrix which can be easily mapped to OpenAI Gym’s Box observation type. However, this mapping introduced additional challenges, which include dealing with invalid actions. In the original environment, the agent always selects a valid edge, because only those are exposed by the graph structure. But now, after the modification, the agent has to differentiate between valid and invalid edges as indicated in the adjacency matrix. To enforce such behavior in each of the frameworks, we propose the following extension, also shown in Figure 4.2. Before an agent picks an action, Q-values for invalid actions returned by the underlying model are replaced with the lowest values possible. As a result, it is guaranteed that the action chosen by the agent (i.e. the one with the highest Q-value) is valid.

4.1.3 Join Order Benchmark

Similarly to the previous research on applying deep reinforcement learning in join order optimization [KYG+18], we use the Join Order Benchmark to evaluate the performance of DRL agents implemented by the frameworks. The benchmark implements 33 query structures with 2-6 modifications that vary only in their selectivity, which results in 113 multi-join queries [LGM+15]. A typical query with a corresponding query graph is shown in Figure 4.3. Solid edges represent a primary-foreign key relationships between tables, while dotted edges represent foreign-foreign key joins.

By relying on the Internet Movie Data Base (IMDB)\(^2\) rather than on a synthetic data set, the benchmark provides diverse and realistic workloads which challenge assumptions of traditional query optimizers[LGM+15]. In this research, IMDB’s version from May 2013 was used\(^3\), which contains 21 tables (3.6 GB in size) with the largest table (cast_info) having 36 million rows.

\(^2\)The license and links to the current version of IMDB dataset can be found at http://www.imdb.com/interfaces
\(^3\)The CSV files for this version of the dataset can be found at http://homepages.cwi.nl/~boncz/job/imdb.tgz
4.2 Configuration of DRL Agents

Depending on the availability, two to four DRL agents from each of the frameworks were picked for the evaluation. From Dopamine framework, we evaluated all of the available algorithms: original Deep Q-Network, Rainbow agent and the Implicit Quantile network. At the time of writing, Horizon has a rather limited set of agents compatible with environments with discrete action spaces. Therefore, only two Horizon agents were evaluated: plain Deep Q-Network and the Rainbow agent. Finally, from the RLlib framework, we picked vanilla Deep Q-Network, the Rainbow agent, the Proximal Policy Optimization agent, and Ape-X version of the DQN algorithm as an example of a highly-distributed agent.

Figure 4.4: Unified neural network for evaluated deep reinforcement learning agents.

All of the aforementioned agents use the same underlying neural network, depicted in Figure 4.4. This simple architecture consists of an input layer which matches the shape of the environment’s observations, two hidden layers with 512 fully-connected neurons and an output layer that matches the size of the environment’s action space. As mentioned in Chapter 2, Rainbow and Implicit Quantile agents differ from the original DQN agent by adding several extensions to the deep neural network. All of the selected frameworks accept the original network and automatically extend it with required components. However, only RLlib creates a final network that completely matches the architecture of the Rainbow agent proposed in the original paper. Both Dopamine and Horizon add only selected components. For instance, Dopamine supports the prioritized replay buffer, n-step and distributional learning, but doesn’t implement double DQN and dueling architectures. Similarly, Horizon only implements the prioritized replay buffer, double DQN and dueling architectures, but doesn’t support the n-step and distributional learning. As a result, in our evaluation, we were able to create comparable neural networks only for the original DQN agent.
As mentioned previously, all of the selected DRL solutions offer configuration tools for a simple initialization of agent’s components such as underlying network, replay buffer, optimization algorithm for the network’s objective function etc. The main focus in this experimental setup was to ensure the identical configuration of the agents across all frameworks. While agent-specific parameters (e.g. \( \gamma \), replay buffer size etc) are equal in all frameworks, the learning workflow of the agent can strongly vary depending on the implementation:

- Agent’s learning process in **Dopamine** is broken down into *timestep-centric* iterations, which means that in one iteration, the agent samples \( n \) experiences, optimizes the network parameters and evaluates the new policy for \( m \) timesteps.

- Although **Ray RLlib** agents also employ the iteration-based workflow, each iteration can be either *episode-centric* or *time-centric*. In the former case, the agent interacts with the environment for \( k \) episodes, optimizes the policy and evaluates it for another \( l \) episodes. Due to the stochasticity of the environment and the agent, these episodes can contain a different number of observations, which results in a different length of a single iteration. In the time-centric setting, a certain time limit is assigned to the training and evaluation phases, which again results in an unpredictable amount of executed timesteps.

- **Horizon** employs a combination of both workflows, where the agent’s learning is split into episodes instead of iterations, and after certain amount of steps, the current episode is paused, the policy is optimized and evaluated, and the episode is resumed.

Since these heterogeneous training workflows make it difficult to compare framework’s runtimes and argue about the efficiency of reference implementations, we unify framework’s learning processes by introducing new parameters (e.g. `agent_training_steps`) and extending agent’s components. As a result, all agents used in this evaluation employed the Dopamine-like iteration-based workflow, whereas each iteration is split into \( n \)

<table>
<thead>
<tr>
<th>Dopamine</th>
<th>Horizon</th>
<th>Ray RLlib</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Used processing units</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>tf_device</code></td>
<td><code>use_gpu</code></td>
<td><code>num_gpus</code></td>
</tr>
<tr>
<td><strong>Name of the registered environment</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>environment_name</code></td>
<td><code>env</code></td>
<td><code>env</code></td>
</tr>
<tr>
<td><strong>Name of the agent to train</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>agent_name</code></td>
<td><code>model_type</code></td>
<td><code>run</code></td>
</tr>
<tr>
<td><strong>Number of iterations for the experiment</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>num_iterations</code></td>
<td><code>via timesteps_total</code></td>
<td><code>training_iteration</code></td>
</tr>
<tr>
<td><strong>Number of training steps per iteration</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>training_steps</code></td>
<td><code>via train_every_ts</code></td>
<td><code>agent_training_steps</code></td>
</tr>
<tr>
<td><strong>Number of evaluation steps per iteration</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>evaluation_steps</code></td>
<td><code>avg_over_num_steps</code></td>
<td><code>agent_evaluation_steps</code></td>
</tr>
<tr>
<td><strong>Maximum number of steps in the episode</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>max_steps_per_episode</code></td>
<td><code>max_steps</code></td>
<td><code>horizon</code></td>
</tr>
</tbody>
</table>

**Table 4.1:** Essential configuration parameters for the DQN-based agents in the evaluated frameworks.
training steps and $m$ evaluation steps. For an overview of these and other essential configuration parameters for the DQN agent, refer to Table 4.1 and Table 4.2. Additionally, in Section A. and Section B., we provide several tuned configurations of the DQN agent for all evaluated frameworks and environments.

<table>
<thead>
<tr>
<th>Dopamine</th>
<th>Horizon</th>
<th>Ray RLlib</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reward discount factor</strong></td>
<td>gamma</td>
<td>gamma</td>
</tr>
<tr>
<td><strong>Probability of choosing random action</strong></td>
<td>epsilon</td>
<td>epsilon</td>
</tr>
<tr>
<td><strong>Learning rate of gradient descent</strong></td>
<td>learning_rate</td>
<td>learning_rate</td>
</tr>
<tr>
<td><strong>The frequency of learning</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>The frequency with which the target network is updated</strong></td>
<td>target_update_period</td>
<td>target_update_rate</td>
</tr>
<tr>
<td><strong>n-step Q-learning</strong></td>
<td>update_horizon</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Min. number of states to observe before training</strong></td>
<td>min_replay_history</td>
<td>via train_after_ts</td>
</tr>
<tr>
<td><strong>Max. number of transitions in experience replay</strong></td>
<td>replay_capacity</td>
<td>max_replay_memory_size</td>
</tr>
<tr>
<td><strong>Number of transitions to sample for training</strong></td>
<td>batch_size</td>
<td>minibatch_size</td>
</tr>
<tr>
<td><strong>Optimization algorithm for the minimization of objective function</strong></td>
<td>optimizer</td>
<td>optimizer</td>
</tr>
<tr>
<td><strong>Frequency of saving model’s state to the disk</strong></td>
<td>checkpoint_freq5</td>
<td>checkpoint_freq5</td>
</tr>
</tbody>
</table>

Table 4.2: Essential configuration parameters for the DQN-based agents in the evaluated frameworks (continuation).

### 4.3 Evaluation Environment

All results reported in this study were obtained for the following versions of the selected deep reinforcement learning packages: Dopamine v.2.0.3, Horizon v.0.1 (commit 2901f36) and Ray v.0.7.0. The Docker image for running experiments was created using Docker v.18.9.6 and nvidia-docker6 v.2.0.3. The image is based on nvidia/cuda:9.0-base-ubuntu16.04 which supports Linux NVIDIA driver versions $\geq 384.81$ and Windows NVIDIA driver versions $\geq 385.54$.

5Configurations for other agents can be found at https://github.com/pshevche/drl-frameworks
5Custom parameters added to simplify the configuration and supported only by the proposed evaluation platform.
6https://github.com/NVIDIA/nvidia-docker
All experiments were computed on a server machine running x86_64 Linux 2.6.32.59, with 32 Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz processors and 256 GB of RAM. Furthermore, the experiment machine had 2 K20X GPUs with a total board memory of 12 GB.

4.4 Summary

In this chapter, we provided essential information about our experimental setup to ensure reproducibility of the results in the future. We started by reviewing two sequential decision making tasks, that we tried to solve using agent implementations from selected frameworks. The first environment, called CartPole, comes from OpenAI Gym benchmark, the well-established collection of environments for evaluation of deep reinforcement learning algorithms. For this task, the goal of the agent is to balance a pole, attached to a cart on a frictionless track, by applying ’left’ or ’right’ impulses. The simplicity of this environment allows us to conduct a large number of experiments with varying parameters, and get a deeper understanding of framework’s properties.

The second task used in our comparison comes from the novel Park benchmark, which contains 12 environments representing a wide range of sequential decision making tasks found in real-world computer systems. In our evaluation, we used the Query Optimizer environment, in which the agent observes different SQL queries involving numerous tables, and has to find an optimal join order for these relations. Queries used for training and evaluation of selected agents come from the Join Order Benchmark based on the Internet Movie Data Base and containing 113 multi-join queries.

Then, in Section 4.2, we determined frameworks’ agents to be used in the evaluation phase of this research. From Dopamine framework, we selected DQN, Rainbow and Implicit Quantile Networks agents. Horizon’s implementations of the first two agents were also considered in our evaluation. Finally, from a wide range of agents supported by the Ray RLlib framework, we picked value-based DQN, Rainbow and Ape-X DQN agents, and policy-based Proximal Policy Optimization algorithm. All these agents had the same underlying deep neural network, consisting of input, output and two hidden layers with 512 neurons. Since the performance of deep reinforcement learning agents heavily depends on the selection of hyperparameters, we enhance the reproducibility of reported results by sharing a complete configuration of evaluated agents in Section A. and Section B.

In the last section of the chapter, we provided some additional technical details regarding the versions of used libraries, as well as complete specifications of utilized hardware resources.
5 Evaluation and Results

In the following chapter, we provide the final details regarding the experimental setup, present results of our empirical evaluation and discuss some of the most valuable observations:

1. In Section 5.1, we define the complete set of experiments conducted in this research, and we discuss how they help us address research questions, presented in previous chapters. Additionally, in this section, we review the varying parameters in each of the experiments and provide our assumptions regarding the future results.

2. Section 5.2 and Section 5.3 are dedicated to the review of obtained empirical results for two selected sequential decision making tasks: CartPole environment from OpenAI Gym benchmark and Query Optimizer environment from Park benchmark.

3. In Section 5.4, we summarize the most essential findings of this comparative evaluation of DRL frameworks and discuss some of the most interesting observations.

5.1 Complete Suite of Conducted Experiments

In the following, we revisit research questions, discussed in Section 3.1, and present the setup of individual experiments that address these questions.

1. **Which of the selected frameworks has the lowest training time and achieves the quickest convergence for a given problem and selected common model?**
   To address this question, we compared frameworks’ implementations of a well-established Deep Q-Network agent against each other on two tasks: a classic control CartPole task from OpenAI Gym benchmark and a real-world query optimization task from the Park benchmark. In contrast to other agents like Rainbow, DQN provides little algorithmic variability, which makes agent’s implementations comparable across different solutions and allows us to focus solely on the framework’s properties and used technologies. For this experiment, we report the total runtime of the learning process and rewards collected during the evaluation phase. All of the results reported below are averaged over several trials with the same agent and environment configurations.

2. **How well can these off-the-shelf solutions utilize different (multiple) processing units?**
   The experiment described previously was run both on a single CPU and on a single GPU. We assume that the PyTorch-based Horizon will achieve the largest speedup when learning on GPU, since this deep learning library has been shown to benefit the most from using graphics processing units [BRSS15].
3. **How does the number of agent’s checkpoints saved to the primary storage during the training process affect the framework’s performance?**

During learning, all of the selected frameworks regularly save the experiment’s state to disk for recovery in case of failures or future re-use of learned weights and collected experiences. By varying the frequency of such saves, we adjust the number of disk accesses and inspect how this parameter influences the runtime of the agent. Similarly to the previous experiments, we evaluated the performance of an original DQN agent on a single CPU. All in all, we ran four experiments with checkpoint frequencies of 0, 0.01, 0.1 and 1. Although the structure of checkpoints varies by the framework (e.g. Dopamine saves both the weights of the network and the content of the experience replay buffer, while Horizon saves only collected experiences), we expect that the increasing checkpoint frequency will result in an increasing runtime for all of the frameworks.

4. **Which of the DRL algorithms implemented by the framework has the best performance for a given problem?**

To address this research question, we selected several reference implementations of common DRL agents and compared their performance both in terms of achieved rewards and total runtime (both on CPU and GPU). From Dopamine, three DQN-based agents were evaluated: original Deep Q-Network, the Rainbow agent and Implicit Quantile Networks. Similarly, Horizon’s implementations of DQN and Rainbow agents were compared against each other. And finally, value-based DQN and Rainbow were compared against a policy-based Proximal Policy Optimization agent in Ray RLlib. Furthermore, to give a first impression of Ray RLlib’s distributed potential, we picked Ape-X variation of DQN agent, as an example of a highly-distributed DRL agent. It is worth mentioning that in this experiment, agents’ implementations were compared only within a single framework. Although all solutions support the Rainbow agent, we don’t find it reasonable to compare the implementation of this agent across frameworks due to the high variability in implemented components.

5. **Once the model has learned, how fast can the framework perform inference (i.e. interact with the environment to solve the given task)?**

One of the key objectives when integrating a DRL agent into a real-world application is to reduce the overhead it has on the production environment. The trained agent deployed to the production system shouldn’t hurt system’s performance and user’s experience, and must be capable of picking millions of actions per second [MNW+18]. Therefore, to determine which framework satisfies this criteria the best, we added the following phase to all of the aforementioned experiments. Once the agent has been trained, we let it interact with the environment for another 1000 timesteps and measure how much time it needs to make a prediction.

### 5.2 Evaluation Results for OpenAI Gym’s CartPole Environment

If not stated otherwise, all of the results presented below are averages over three trials, whereas each trial consisted of 500 iterations with 500 training and 1000 evaluation steps.
per iteration. The agent and environment configurations remained constant across the trials and frameworks, and can be found either in Section A. or in the project’s repository\textsuperscript{1}.

**RQ1: Which of the selected frameworks has the lowest training time and achieves the quickest convergence for a given problem and selected common model?**

As we can see in Figure 5.1, Ray RLlib’s implementation of the DQN agent achieves the best performance both in terms of the runtime and achieved rewards\textsuperscript{2}. In each of the trials, this implementation of the agent was able to find the policy producing the maximal reward (i.e. 200 for the CartPole environment) within the first 50 iterations (i.e. 11 minutes in terms of runtime). Both Horizon’s and Dopamine’s implementations of the agent showed high variation of reward values across different trials. Although Horizon reported similar results in each of the runs, it was able to find the optimal policy only in one of three trials. An interesting behavior of the agent was observed for the Dopamine’s implementation of the approach. The agent was able to quickly find the solution of the problem (in each of the trials, the optimal policy was obtained within the first 20 iterations), but degraded to the sub-optimal solution just as quickly (by the 50th iteration, two out of three trials reported an average testing reward of 10).

![Figure 5.1: Average rewards per evaluation episode (left) and the total duration of the learning process (right) for the frameworks’ implementations of a DQN agent and OpenAI Gym’s CartPole environment.](image)

In terms of performance, the Ray RLlib agent trained on CPU halves the runtime of the Dopamine’s implementation and outperforms the Horizon’s implementation by a half-order of magnitude. To explain this behavior and identify components responsible for such results, we conducted a micro evaluation, in which we profiled the runtimes of the agents during the first five iterations. Obtained results are presented in Figure 5.2.

As expected, all agents spend the majority of time optimizing the policy and performing gradient descent steps on the network weights. Since Ray RLlib’s optimizer, responsible for the update of the policy graph, delegates the computation of the gradients to

\textsuperscript{1}https://github.com/pshevche/drl-frameworks/tree/master/experiments/cartpole

\textsuperscript{2}In our evaluation, we report testing rewards per iteration of agents trained on GPU.
remote evaluators, running in parallel threads, it is able to perform the training faster, which results in lower total runtimes. Another interesting observation to be made is that TensorFlow-based Dopamine performs the training phase faster than PyTorch-based Horizon, which differs from similar evaluation results from other research, as reported in [BRSS15].

RQ2: How well can these off-the-shelf solutions utilize different (multiple) processing units?

Results reported in Figure 5.1 and Figure 5.2 confirm our initial assumption that the PyTorch-based Horizon achieves the largest speedup when training an DQN agent on GPU. Already after the first five iterations, GPU-trained DQN agent spends almost 50% less time in the training phase, which decreased the total runtime by a factor of three. Interestingly, TensorFlow-based Dopamine and Ray RLlib don’t show any significant speedup when running on GPUs. Due to the lack of empirical evidence, we can only speculate about the causes of such behavior. Either the underlying fully-connected neural network doesn’t provide much potential for parallelism, or the implementation of our experimental setup has a flaw preventing TensorFlow from running efficiently on the GPU. Although collected execution logs indicate that both Dopamine and Ray RLlib were able to schedule tasks on GPU, we can’t completely exclude the possibility that the graphics processing unit wasn’t used to its full potential.

RQ3: How does the number of agent’s checkpoints saved to the primary storage during the training process affect the framework’s performance?

Our assumption, that the increasing number of disk accesses should result in a decreasing performance, is confirmed by the results shown in Figure 5.3. Even though some cases (e.g. Dopamine DQN with no checkpointing and Horizon DQN with 10% checkpointing) don’t fit the expected performance pattern, in general, we observe a slight increase in runtime with a growing number of accesses to the primary storage.
At first glance, this parameter doesn’t seem to have a significant impact on the performance of the framework, as all of the selected solutions spend less than 1% of the time managing checkpoints. However, we believe that such behavior is caused by a relatively small size of a checkpoint which depends on the representation of environment’s states, actions, rewards and the complexity of underlying neural network. Therefore, we can expect a larger influence of checkpointing in more complex tasks, where environment’s components may be represented by thousands of values. Finally, we’d like to point out the checkpointing strategy employed by the Horizon framework. In contrast to Dopamine and Ray Rllib, Horizon doesn’t store the complete state of the agent after each iteration (which explains the low size of a single checkpoint), but accumulates collected experiences and stores them in one go at the end of the learning process. Such setup doesn’t support a smooth recovery in case of failures and requires millions of transitions to be persisted in main memory. The impact of this checkpointing strategy remains to be evaluated in future work.

**RQ4: Which of the DRL algorithms implemented by the framework has the best performance for a given problem?**

The setup of this experiment is very similar to the evaluation that helped us address RQ1. But instead of comparing the performance of a single agent across multiple frameworks, in this experiment, we compare the performance of multiple agents within a single framework. Figure 5.4 provides an overview of obtained results.

In Dopamine framework, only the Implicit Quantile Networks agent was able to find the optimal policy for the CartPole environment. Although previous results have shown that the Rainbow agent tends to perform better than IQN [DOSM18], in our evaluation we observe the opposite behavior. Since the Rainbow agent performed poorly in all of the frameworks, we assume that the agent suffers from inadequately chosen hyperparameters. Therefore, in future work, a significant effort has to be put into finding configurations that fully exploit the benefits of the agents. In terms of runtime, we expect Rainbow and IQN to be slower than DQN due to a more sophisticated architecture of the neural networks (e.g. Rainbow has an additional layer that represents the distribution of Q-values) and
additional computations (e.g. IQN agent integrates additional functions for the quantile regression).

As mentioned previously, Horizon framework offers a limited set of agents compatible with environments with discrete state and action domains. Therefore, in this research, we only compared the performance of DQN and Rainbow implementations. Similarly to the results obtained for the previous framework, Horizon’s implementation of the Rainbow agent is slower than DQN and delivers poor results, which confirms our assumption about the improper configuration of the approach. Furthermore, in contrast to the Dopamine and Ray RLlib implementations, all Horizon agents can benefit in terms of runtime when training is performed on the GPU.

Finally, in the Ray RLlib framework, all approaches apart from the Rainbow agent were able to find the optimal policy, while two of them: PPO and Ape-X DQN, show the most stable behavior. Furthermore, the PPO agent showed superior performance in terms of runtime and outperformed the Rainbow and Ape-X DQN agents by more than one order of magnitude. The main reason behind it is that PPO used multiple evaluators distributed over several CPUs for the collection of experiences and the computation of gradients. Although this setup may seem to be unfair towards DQN and Rainbow agents that run on a single processing unit, we tried to reproduce conditions most likely to be...
used in production setting. And as discussed previously, to ensure sample efficiency of the PPO agent, it is common to have multiple CPU-based workers collecting experiences and a single GPU-based optimizer for updating the policy. Ape-X variation of the DQN agent works in a similar setting, where a single GPU learner is used to optimize the policy and multiple CPU workers interact with the environment. However, in contrast to the PPO agent, each worker can have a different version of the policy to collect experiences and fill the replay buffer. Therefore, Ape-X DQN employs the distributed prioritization of experiences to decide which transitions have to be stored in the replay buffer first. Finding empirical evidence explaining the poor runtime of this highly distributed agent remains to be done in the future. Currently, we can only speculate that for such simple environment like CartPole, the overhead for maintaining multiple asynchronous workers is too large and it is more beneficial to use a simple DQN agent on a single worker instead.

**RQ5: Once the model has learned, how fast can the framework perform inference (i.e. interact with the environment to solve the given task)?**

![Figure 5.5: Average time required by the agent to perform an action in the CartPole environment.](image)

To determine which agent and which framework introduce the smallest overhead when deployed in the production environment, we let the trained agent interact with the environment for 1000 steps, and compute the average time needed to select an action. As we see from the results shown in Figure 5.5, Horizon’s implementation of DQN agent is the fastest to make a prediction. All Dopamine’s agents achieve high inference and are capable of performing around 900-1400 actions per second. On the other hand, Ray RLlib’s agents are able to make 800 predictions per second at most. This is mainly caused by a large number of sub-tasks performed during the inference: the Ray RLlib’s policy evaluator not only returns the action to be performed in the observed state, but also coordinates all workers interacting with the environment. It keeps track of all currently executed episodes, preprocesses collected observations, persists statistics per agent, environment and episode, postprocesses obtained rewards and records performance events. Although these steps are essential to ensure distributed learning over multiple nodes,
for a simple case evaluated in this work, it may introduce additional unnecessary overheads.

5.3 Evaluation Results for Park’s Query Optimizer Environment

Due to the complexity of the Park’s Query Optimizer environment and resource limitations on this work, we were not able to run the complete suite of the experiments presented previously. As a result, in the following we present empirical results only for RQ1, RQ4 and RQ5. For the same reasons, all of the presented results are averaged over just two trials, while each trial consisted of 1000 iterations with 300 training and 500 evaluation steps. All of the evaluated agents were trained on GPUs, and no checkpoints were saved during the learning process. Similar to the experiments on the CartPole environment, agent and environment configurations remained constant across the trials and frameworks, and are presented in Section B. and project’s repository.

RQ1: Which of the selected frameworks has the lowest training time and achieves the quickest convergence for a given problem and selected common model?

![Figure 5.6: Average rewards per evaluation episode (left) and the total duration of the learning process (right) for the frameworks’ implementations of the DQN agent and Park’s Query Optimizer environment.](image)

The evaluation results, shown in Figure 5.6, don’t allow us to clearly identify the superior implementation for the interaction with Park’s Query Optimizer environment. All implementations of the DQN agent show oscillating behavior and report similar testing rewards in range between six and eight. However, we assume that such high variance in the observed rewards is mainly caused by the specification of environment’s reward signal and properties of used benchmark queries.

As mentioned previously, each episode of the Query Optimizer environment reports its cumulative reward, while the rewards for individual steps can range from 0 to 1, where 1 is the optimal value. Consequently, the optimal reward per episode is equal to its length,

[^3]: https://github.com/psheche/drl-frameworks/tree/master/experiments/query_optimizer
which depends on the number of relations joined in the query. Since the benchmark queries used in this evaluation perform eight joins on average, the optimal policy should converge against the average reward of eight per evaluation episode. However, due to different types of queries being evaluated after each training iteration, the average reward per episode may sometimes be less and sometimes more than eight. As a result, the oscillating curve observed during this experiment is expected.

To reason about the quality of discovered policy, we have to look at two properties of the evaluation curve: how high is the variance in collected rewards and how fast does the agent approach the average optimal reward. As to be seen from the left chart of Figure 5.6, Dopamine’s and Ray RLlib’s implementations of the DQN agent have similar variance in the collected rewards, while Horizon’s implementation reports a higher distribution of rewards. To discover the trend in agent’s convergence, we calculate the exponential moving average [LL77] on the series of evaluation rewards and plot the smoothed signal in Figure 5.7 (the degree of weighting decrease \( \alpha \) is equal to 0.01). As we can see, all agents slowly approach the optimal reward, while Ray RLlib’s version of the agent slightly outperforms the remaining two implementations.
Similarly to the CartPole experiments, Ray RLlib’s implementation of the DQN agent shows the best performance in terms of runtime. In contrast to the previous results, Horizon’s implementation of the GPU-based agent showed poor performance and exceeded the available memory limit after 747 iterations. Therefore, in the right part of Figure 5.6, we report both the actual runtime for completed iterations and the estimated final performance. Profiling results, shown in Figure 5.8, indicate that Horizon’s DQN agent required more time for training than its competitors. Interestingly, the agent has spent the majority of the training phase (appr. 81% of time) sampling transitions from the experience replay buffer. Clarifying this behavior for such complex environments like Park’s Query Optimizer should be a crucial aspect of future research.

RQ4: Which of the DRL algorithms implemented by the framework has the best performance for a given problem?

Figure 5.9: Smoothed average rewards per evaluation episode (right) and the total duration of the learning process (left) for the frameworks’ implementations of several DRL agents and Park’s Query Optimizer environment.

Figure 5.9 summarizes the empirical results obtained for this experiment. For the same reasons as previously, we report the exponential moving average of the reward signal instead of raw rewards (the original signal can be found in Section C.). In this experiment, we don’t observe any significant difference in the quality of agents’ policies. All of the evaluated DRL approaches show similar oscillating behavior and slow but incremental
convergence towards the optimal reward. However, some interesting patterns can be discovered in the total duration of the learning process of various agents. In contrast to the CartPole environment, Implicit Quantile Networks outperformed the Rainbow agent while achieving similar rewards. We argue that Rainbow’s poor performance is caused by a large increase in the size of environment’s action space (1681 for Query Optimizer environment against 2 for CartPole environment). A larger number of actions also results in a larger size of the distributional layer in Rainbow’s deep neural network, which has a negative impact on the agent’s performance. Since both Horizon agents failed to complete the training, we won’t discuss their performance in details and leave new evaluations to future research. Finally, similarly to the Dopamine’s implementation of the approach, Ray RLlib’s version of the Rainbow agent showed the worst performance while interacting with the Query Optimizer environment. On the other hand, Proximal Policy Optimization further establishes its superiority and outperforms DQN and Rainbow agents by factors of 1.5 and 14 respectively.

**Figure 5.10:** Average time required by the agent to perform an action in the Query Optimizer environment.

**RQ5:** Once the model has learned, how fast can the framework perform inference (i.e. interact with the environment to solve the given task)?

Since Horizon agents couldn’t complete the training, we did not consider it appropriate to perform inference testing for this framework. Therefore, in the following, we compare only inference results for agents implemented in Dopamine and Ray RLlib. As we see in Figure 5.10, for Query Optimizer environment, the Ray RLlib’s implementations were able to achieve better inference than Dopamine agents. Profiling results, shown in Figure 5.8, indicate that although both frameworks can make predictions equally fast (appr. 5ms to perform the forward pass through the network), Dopamine requires more time to receive and process environment’s feedback (appr. 7ms for Dopamine against 5ms for Ray). And exactly this difference of 2ms per action can be observed in the presented chart. Even though we don’t have the break down of Rainbow’s inference times, we can expect similar behavior for implementations of this agent.
5.4 Summary

In this chapter, we presented the complete suite of experiments conducted in the evaluation phase of our research and reviewed some of the most valuable results. Firstly, we attempted to determine which framework provides a design that ensures the best performance both in terms of runtime and collected rewards. We trained framework’s versions of traditional Deep Q-Network on two environments: OpenAI Gym’s CartPole and Park’s Query Optimizer. While Ray RLlib’s implementation of the approach showed superior performance for both tasks, PyTorch-based Horizon achieved the maximal relative speedup when training was done on GPU. Secondly, the impact of the checkpointing frequency on the total duration of training was studied. Although the increasing number of stores to the disk causes just a slight decrease in frameworks’ performance, we assume that this parameter may have a larger effect on agent’s performance when interacting with more complex environments. Thirdly, we compared the performance of multiple DRL agents within individual frameworks and found out that Dopamine’s Implicit Quantile Networks, Horizon’s DQN and Ray RLlib’s Proximal Policy Optimization achieve the best results both in terms of runtime and observed rewards. Finally, once the training was finished, we let the trained agent interact with the environment for additional 1000 steps and measured how fast can the agent perform a single action in this environment. We observed different results for CartPole and Query Optimizer environments, which indicates that the inference time depends not only on the framework’s design but also on the implementation of environment’s feedback mechanism. While Dopamine versions of DRL approaches reported better inference times for the CartPole task, Ray RLlib agents performed better on a more complex Query Optimizer environment.
6 Related Work

1. **Comparison of deep reinforcement learning frameworks.**
   To our knowledge, this work is currently the only empirical evaluation of deep reinforcement learning solutions. Previous comparisons were limited to outlining the differences in frameworks’ purposes and components, and were conducted by teams introducing new frameworks [GCL+18][CMG+18][GDK+15]. However, parts of this research were motivated by the earlier evaluation of commonly used deep learning frameworks like Caffe, Neon, TensorFlow, Theano and Torch [BRSS15]. Similarly to Bahrampour et al., we investigate the influence of various configuration parameters on the performance of the framework and the ability of the solution to utilize different (multiple) processing units. In contrast to our research, [BRSS15] puts focus on the capability of the deep learning frameworks to incorporate different types of neural networks, training procedures and backend libraries.

2. **Application of DRL algorithms in production environments.**
   While early applications of deep reinforcement learning were mainly confined to controlled environments, the abundance of production data and large amount of use cases have motivated numerous research in applying DRL techniques in real-world computer systems ([LNB+18], [ZRF+18], [RK17]). Previous research, for instance, has also tackled the issue of applying deep reinforcement learning techniques in the domain of query optimization [KYG+18][MNM+19]. Although their findings motivated some design decisions of our evaluation setup, in this work we mainly focus on testing several off-the-shelf DRL agents on the join order optimization task, rather than developing task-specific agents and integrating them deep into existing database management systems.
   In addition to providing a flexible infrastructure for connecting frameworks to a running instance of a Postgres database, the Park benchmark also includes reference training results of the DQN agent for the query optimization environment. Similarly to the previously discussed works, Park project relied on its own implementation of the agent rather than some commonly used reference implementation. Since implementing such agents is a sophisticated task, requiring deep understanding of the domain, a development team would rather use or extend an existing solution and apply it to a domain-specific problem. Therefore, in this research, we evaluated the applicability of several well-established implementations of DRL agents for the query optimization environment provided by the Park benchmark.
7 Conclusion and Future Work

In the following, we summarize the main contributions of this research, discuss some possible improvements to the presented evaluation setup and outline some promising directions for future research.

7.1 Conclusion

To our knowledge, this research is the first attempt to empirically compare existing frameworks that provide deep reinforcement learning functionality. In this work, we presented a wide range of state-of-the-art solutions, three of which were selected for an extensive evaluation: research-oriented Dopamine, an end-to-end production platform Horizon and Ray RLlib that enhances distributed learning of DRL agents.

Firstly, we tried to determine a framework with a design that provides the best performance both in terms of runtime and observed rewards. By comparing frameworks’ implementations of a well-established DQN agent, we found out that Ray RLlib’s parallel abstract components satisfy these requirements the best.

Secondly, we trained several DRL agents both on CPU and GPU, and discovered that Horizon’s implementations based on the PyTorch deep learning library achieve the highest speedup when run on graphics processing units (69% in comparison to Dopamine’s 2.5% and Ray RLlib’s 0.3%). This, however, does not make the agents of Horizon more competitive than agents of other frameworks.

Thirdly, the impact of checkpointing frequency on the framework’s performance was evaluated. Experiment’s results confirmed our assumption that the increasing frequency of saves decreases framework’s performance and may be an important factor when interacting with complex environments. All frameworks experienced a runtime increase of 2-4% when agent’s state was stored to the disk after each iteration.

Fourthly, we conducted a series of intra-framework experiments, in which we tried to determine which of the framework’s agents deliver the best results for two benchmark environments: OpenAI Gym’s CartPole and Park’s Query Optimizer environments. Dopamine’s Implicit Quantile Networks, Horizon’s Deep Q-Network and Ray RLlib’s Proximal Policy Optimization agents strike the perfect balance between runtime and policy’s quality for both problems.

Finally, for a variety of agents, we measured inference times, which is a key parameter to consider when deploying framework’s policies to production systems. For this evaluation, we found out that Dopamine’s agents achieve high inference for simple environments like CartPole (0.7ms per single interaction), while Ray RLlib’s sophisticated monitoring system is beneficial for more complex problems like Park’s Query Optimizer (8ms per...
action). These and other properties of the evaluated frameworks are summarized in Table 7.1.

<table>
<thead>
<tr>
<th>Property</th>
<th>Dopamine</th>
<th>Horizon</th>
<th>Ray RLlib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework’s focus</td>
<td>Research Education</td>
<td>Production</td>
<td>Distributed learning</td>
</tr>
<tr>
<td>Environments</td>
<td>OpenAI Gym User-defined</td>
<td>OpenAI Gym User-defined</td>
<td>OpenAI Gym User-defined</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ELF</td>
</tr>
<tr>
<td>Agents</td>
<td>Value-based</td>
<td>Value-based</td>
<td>Value-based</td>
</tr>
<tr>
<td>Setting</td>
<td>Online</td>
<td>Online</td>
<td>Online</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Offline</td>
<td>Offline</td>
</tr>
<tr>
<td>Supported DL libraries</td>
<td>TensorFlow</td>
<td>PyTorch</td>
<td>PyTorch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TensorFlow</td>
</tr>
<tr>
<td>GPU support</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed learning</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Code base (LoC)</td>
<td>35 files (4798)</td>
<td>126 files (23,434)</td>
<td>2k files (590,924)</td>
</tr>
<tr>
<td>DQN implementation(^1)</td>
<td>Convergence: ✗ Runtime rank: 3</td>
<td>Convergence: ✗ Runtime rank: 2</td>
<td>Convergence: ✓ Runtime rank: 1</td>
</tr>
<tr>
<td>GPU speedup</td>
<td>2.54%</td>
<td>69.4%</td>
<td>0.3%</td>
</tr>
<tr>
<td>Runtime increase (0% to 100% saves)</td>
<td>1.9%</td>
<td>4%</td>
<td>3.9%</td>
</tr>
<tr>
<td>Most performant implementation</td>
<td>IQN</td>
<td>DQN</td>
<td>PPO</td>
</tr>
<tr>
<td>Number of actions per second (DQN)</td>
<td>CartPole: 1257 QOpt: 97</td>
<td>CartPole: 1385 QOpt: N/A</td>
<td>CartPole: 726 QOpt: 125</td>
</tr>
</tbody>
</table>

**Table 7.1:** Summary of framework’s properties and evaluation results for Dopamine, Horizon and Ray RLlib.

### 7.2 Threats to Validity

In the following, we summarize some of the problems that we encountered during our evaluation and how they influenced the conclusions of our work.

- **Effect of hyperparameters and network architecture.**
  Previous research has shown that DRL agents are very sensitive to the choice of hyperparameters and underlying network architecture [HIB\(^+\)18]. In our evaluation, we established a configuration that showed satisfying results on a small set of evaluations, and replicated it across frameworks for the complete suite of experiments. All our conclusions regarding the stability of implementation’s learning process are

\(^1\)Our conclusions about the implementation’s quality are based on the empirical results obtained for the selected common hyperparameter setting and network architecture.
based on results obtained for this common configuration and defined unified network architecture. We acknowledge the fact, that by using additional expertise and common hyperparameter tuning algorithms \cite{LJR18, JDO17}, we could find a configuration that ensures a different, potentially better, performance of frameworks in terms of collected rewards.

- **Configuration of the Rainbow agent.**
  Numerous previous evaluations have reported Rainbow’s state-of-the-art performance on a variety of sequential decision making tasks. However, in our experiments, we observed a rather poor performance of the agent even for such simple tasks like OpenAI Gym’s CartPole environment. We assume that this behavior is caused by an inadequate selection of agent’s hyperparameters, and finding a proper configuration can change our recommendations of frameworks’ implementations.

- **Park Query Optimizer’s observation space.**
  To make Park’s Query Optimizer environment compatible with selected frameworks, we implemented a wrapper that transforms environment’s graph observations into adjacency matrices. Although from the theoretical point of view these two representations are equivalent, the matrix representation induced additional overhead for the agents (e.g. need to perform action pruning) and couldn’t include all of the graph’s properties (e.g. edges’ and nodes’ features). The native support for graph observation spaces within the selected frameworks could change the performance results reported in this evaluation.

- **Dopamine’s and Ray RLlib’s GPU performance.**
  In this evaluation we observed that TensorFlow-based frameworks Dopamine and Ray RLlib showed small improvements when used with GPUs, for the networks evaluated, which contradicts results reported in previous research \cite{BRSS15}. Although collected execution logs indicate that both frameworks successfully accessed and scheduled tasks on GPUs, we can’t completely exclude a possibility of an implementation flaw in our experimental setup.

- **Horizon’s memory limit violation.**
  During the experiments on Park’s Query Optimizer environment, we observed that Horizon’s agents quickly exceed the available memory limit and fail to complete the training. This prevented us from obtaining results for a series of experiments, and, as a result, makes our conclusions for the Query Optimizer environment only partially valid. Once the flaw is fixed and missing results are obtained, we can fully reason about the performance of this framework in comparison to selected competitors.

### 7.3 Future Work

Comparative evaluations like ours assist researchers and developers in picking a solution that fits their needs. In this research, we investigated the impact of several parameters on the performance of a limited number of DRL frameworks. We recommend the research community to elaborate on our contributions by addressing the following questions in future works.
- **Extend current evaluation to cover a wider range of frameworks.**
  As mentioned previously, recent developments in the field of deep reinforcement learning caused a significant growth in a number of solutions that provide reference implementations of DRL approaches. Future evaluations of such promising frameworks like Amazon Sagemaker RL, Intel Coach and PARL are key extensions to our work, as they can further improve our understanding of a heterogeneous landscape of existing solutions and their design choices.

- **Evaluation of frameworks’ safety and usability aspects.**
  In this work, we mainly focused on the performance of framework’s implementations in terms of runtime and collected rewards. However, the decision to apply a certain solution in a production or research setting can’t be based solely on these aspects. Firstly, collecting empirical evidence about safety properties of DRL frameworks is a crucial next step, which will not only encourage the usage of RL methods in safety-critical applications, but can also help us dispel common misconceptions about AI’s nature. A good starting point would be to extend our evaluation by adding experiments on the AI Safety Gridworld benchmark, presented in one of the previous chapters.
  Secondly, adding a new solution to the company’s technology stack can significantly slow down the development process. Therefore, prior to adopting existing DRL frameworks in production, we should be able to reason about solution’s maintainability and the quality of its code base. Previous studies in the field of web development provide examples of several code metrics that can be used to reason about framework’s usability [GCP12].

- **Exhaustive performance evaluation for real-world environments.**
  With an introduction of the Park benchmark, it became possible to evaluate DRL algorithms on a wide range of representative real-world sequential decision making tasks. In this evaluation, we used just one environment from this suite, which, however, helped us instantly discover several weaknesses in frameworks’ designs (e.g. Horizon’s memory issues). We believe that adapting our experiments to further environments will allow us to detect further imperfections of existing frameworks and push their development process in the right direction.

- **Dedicated study for learning in offline setting.**
  As mentioned in Section 2.1, deep reinforcement learning approaches can interact with the environment in two settings. Previous algorithmic research, as well as our evaluation, compared agents in an online setting. However, this strategy is very unlikely to be used in production, where the learning agent can negatively influence system’s performance and user’s experience. Therefore, there is an obvious need for a comparative study that will define the best practices for evaluation in offline setting and provide baseline results for existing implementations of DRL agents.

- **Cross-platform standardization.**
  Even though the heterogeneity of frameworks is essential to support a wide range of diverse DRL tasks, it makes the analysis of agents’ performance extremely challenging. Developing guidelines for such parts of the DRL process like configuration of the agents, reporting of training metrics and checkpointing ensures a better reproducibility of learning workflows across frameworks. This standardization will not
only simplify the evaluations of software solutions, but will also pose large benefits for the algorithmic research dealing with fundamental properties of DRL agents.
Bibliography


[SSSS16] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. arXiv preprint arXiv:1610.03295, 2016.


Appendices

A. Example Configurations of the DQN Agent for the CartPole Environment.

Configuration of Dopamine’s DQN agent for the CartPole environment.

```python
import dopamine.discrete_domains.gym_lib
import dopamine.discrete_domains.run_experiment
import dopamine.agents.dqn.dqn_agent
import dopamine.replay_memory.circular_replay_buffer
import gin.tf.external_configurables

DQNAgent.observation_shape = %gym_lib.CARTPOLE_OBSERVATION_SHAPE
DQNAgent.observation_dtype = %gym_lib.CARTPOLE_OBSERVATION_DTYPE
DQNAgent.stack_size = %gym_lib.CARTPOLE_STACK_SIZE
DQNAgent.network = @gym_lib.cartpole_dqn_network
DQNAgent.gamma = 0.99
DQNAgent.update_horizon = 1
DQNAgent.min_replay_history = 500
DQNAgent.update_period = 1
DQNAgent.target_update_period = 100
DQNAgent.epsilon_fn = @dqn_agent.identity_epsilon
DQNAgent.tf_device = '/gpu:0' # use '/cpu:*' for non-GPU version
DQNAgent.optimizer = @tf.train.AdamOptimizer()

tf.train.AdamOptimizer.learning_rate = 0.0001
tf.train.AdamOptimizer.epsilon = 0.0003125

create_gym_environment.environment_name = 'CartPole'
create_gym_environment.version = 'v0'
create_parametric_agent.agent_name = 'dqn'
CheckpointRunner.create_environment_fn = @gym_lib.create_gym_environment
CheckpointRunner.num_iterations = 500
CheckpointRunner.training_steps = 1000
CheckpointRunner.evaluation_steps = 1000
CheckpointRunner.max_steps_per_episode = 200
CheckpointRunner.checkpoint_freq = 0
CheckpointRunner.inference_steps = 1000

WrappedReplayBuffer.replay_capacity = 50000
WrappedReplayBuffer.batch_size = 128
```
Configuration of Horizon’s DQN agent for the CartPole environment.

```
"env": "CartPole-v0",
"model_type": "pytorch_discrete_dqn",
"max_replay_memory_size": 50000,
"use_gpu": true,
"rl": {
    "gamma": 0.99,
    "target_update_rate": 0.1,
    "maxq_learning": 1,
    "epsilon": 0.0003125,
    "temperature": 0.35,
    "softmax_policy": 0
},
"rainbow": {
    "double_q_learning": false,
    "dueling_architecture": false
},
"training": {
    "layers": [-1, 512, 512, -1],
    "activations": ["relu", "relu", "linear"],
    "minibatch_size": 128,
    "learning_rate": 0.0001,
    "optimizer": "ADAM",
    "lr_decay": 0.999
},
"run_details": {
    "num_episodes": 1000000,
    "max_steps": 200,
    "train_every_ts": 1,
    "train_after_ts": 500,
    "test_every_ts": 1000,
    "test_after_ts": 1,
    "num_train_batches": 1,
    "avg_over_num_episodes": 100,
    "avg_over_num_steps": 1000,
    "offline_train_epochs": 30,
    "timesteps_total": 500000,
    "checkpoint_after_ts": 0,
    "num_inference_steps": 1000
```
Configuration of Ray’s DQN agent for the CartPole environment.

```python
ray_dqn_gpu_cp0:
  run: DQN
  env: CartPole-v0
  local_dir: "results/cartpole"
  checkpoint_freq: 0
  checkpoint_at_end: False
  agent_training_steps: 1000
  agent_evaluation_steps: 1000
  inference_steps: 1000
  stop:
    training_iteration: 500
  config:
    adam_epsilon: 0.0003125
    buffer_size: 50000
    dueling: False
    double_q: False
    evaluation_interval: 1000000
    learning_starts: 500
    hiddens: [512, 512]
    lr: 0.0001
    num_cpus_for_driver: 0
    num_gpus: 1
    num_workers: 0
    prioritized_replay: False
    train_batch_size: 128
```
B. Example Configurations of the DQN Agent for the Query Optimizer Environment.

Configuration of Dopamine’s DQN agent for the Query Optimizer environment.

```python
import dopamine.discrete_domains.gym_lib
import dopamine.discrete_domains.run_experiment
import dopamine.agents.dqn.dqn_agent
import dopamine.replay_memory.circular_replay_buffer
import gin.tf.external_configurables
import drl_fw.dopamine.components.park_networks

ParametricDQNAgent.observation_shape = (1681,)
ParametricDQNAgent.network = @park_networks.qopt_dqn_network
ParametricDQNAgent.gamma = 0.99
ParametricDQNAgent.update_horizon = 1
ParametricDQNAgent.min_replay_history = 1000
ParametricDQNAgent.update_period = 1
ParametricDQNAgent.target_update_period = 100
ParametricDQNAgent.epsilon_fn = @dqn_agent.identity_epsilon
ParametricDQNAgent.tf_device = '/gpu:0'
ParametricDQNAgent.optimizer = @tf.train.AdamOptimizer()

tf.train.AdamOptimizer.learning_rate = 0.0001
tf.train.AdamOptimizer.epsilon = 0.0003125

create_gym_environment.environment_name = 'ParkQOptEnv'
create_gym_environment.version = 'v0'
create_parametric_agent.agent_name = 'parametric_dqn'
CheckpointRunner.create_environment_fn = @gym_lib.create_gym_environment
CheckpointRunner.num_iterations = 1000
CheckpointRunner.training_steps = 300
CheckpointRunner.evaluation_steps = 500
CheckpointRunner.max_steps_per_episode = 25
CheckpointRunner.checkpoint_freq = 0
CheckpointRunner.inference_steps = 1000

WrappedReplayBuffer.replay_capacity = 50000
WrappedReplayBuffer.batch_size = 128
```
Configuration of Horizon’s DQN agent for the Query Optimizer environment.

```json
"env": "Park00ptEnv-v0",
"model_type": "pytorch_discrete_dqn",
"max_replay_memory_size": 50000,
"use_gpu": true,
"rl": {
  "gamma": 0.99,
  "target_update_rate": 0.33,
  "maxq_learning": 1,
  "epsilon": 0.0003125,
  "temperature": 0.35,
  "softmax_policy": 1
},
"rainbow": {
  "double_q_learning": false,
  "dueling_architecture": false
},
"training": {
  "layers": [-1, 512, 512, -1],
  "activations": ["relu", "relu", "linear"],
  "minibatch_size": 128,
  "learning_rate": 0.0001,
  "optimizer": "ADAM",
  "lr_decay": 0.999
},
"run_details": {
  "num_episodes": 1e6,
  "max_steps": 200,
  "train_every_ts": 1,
  "train_after_ts": 1000,
  "test_every_ts": 300,
  "test_after_ts": 1,
  "num_train_batches": 1,
  "avg_over_num_episodes": 100,
  "avg_over_num_steps": 500,
  "offline_train_epochs": 30,
  "timesteps_total": 300000,
  "checkpoint_after_ts": 0,
  "num_inference_steps": 1000
}
```
Configuration of Ray's DQN agent for the Query Optimizer environment.

```python
ray_dqn_gpu_cp0:
  run: DQN
  env: RayParkQOptEnv-v0
  local_dir: "results/query_optimizer"
  checkpoint_freq: 0
  checkpoint_at_end: False
  agent_training_steps: 300
  agent_evaluation_steps: 500
  inference_steps: 1000
  stop:
    training_iteration: 1000
  config:
    adam_epsilon: 0.0003125
    buffer_size: 50000
    dueling: False
    double_q: False
    evaluation_interval: 1000000
    learning_starts: 1000
    lr: 0.0001
    num_cpus_for_driver: 0
    num_gpus: 1
    num_workers: 0
    prioritized_replay: False
    train_batch_size: 128
    target_network_update_freq: 100
    hiddens: []
  model:
    custom_model: parametric_dqn_model
```
C. Unsmoothed Query Optimizer Rewards Observed by Evaluated DRL Agents.

Average Query Optimizer rewards observed by Dopamine agents.

Average Query Optimizer rewards observed by Horizon agents.

Average Query Optimizer rewards observed by Ray RLlib agents.