Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Distributed Deep Reinforcement Learning for Learn-To-Optimize

Paul Mayer

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Dirk Pflüger
Supervisor:	Peter Domanski, M.Sc.
Commenced:	December 1, 2022
Completed:	June 1, 2023

Abstract

In the context of increasingly complex applications, e.g., robust performance tuning in Integrated Circuit Design, conventional optimization methods have difficulties in achieving satisfactory results while keeping to a limited time budget. Therefore, learning optimization algorithms becomes more and more interesting, replacing the established way of hand-crafting or tweaking algorithms. Learned algorithms reduce the amount of assumptions and expert knowledge necessary to create state-of-the-art solvers by decreasing the need of hand-crafting heuristics and hyper-parameter tuning. First advancements using Reinforcement Learning have shown great success in outperforming typical zeroth- and first-order optimization algorithms, especially with respect to generalization capabilities. However, training still is very time consuming. Especially challenging is training models on functions with free parameters. Changing these parameters (that could represent, e.g., conditions in a real world example) affects the underlying objective function. Robust solutions therefore depend on thorough sampling, which tends to be the bottleneck considering time consumption. In this thesis we identified the runtime bottleneck of the Reinforcement Learning Algorithm and were able to decrease runtime drastically by distributing data collection. Additionally, we studied the effects of combining sampling strategies in regards to generalization capabilities of the learned algorithm.

Kurzfassung

Der Anstieg an Komplexität in modernen Anwendungen, z.B. die Leistungsoptimierung beim Entwurf integrierter Schaltungen, bereitet herkömmlichen Optimierungsalgorithmen Schwierigkeiten, zufriedenstellende Ergebnisse zu erzielen und gleichzeitig ein begrenztes Zeitbudget einzuhalten. Daher wird das Lernen von Optimierungsalgorithmen immer interessanter und ersetzt die etablierte Methode der manuellen Entwicklung und Optimierung dieser Algorithmen. Gelernte Algorithmen reduzieren die Menge an Annahmen und Expertise, die für die Entwicklung von hochmodernen Optimierern erforderlich sind, indem sie die Notwendigkeit der manuellen Erstellung von Heuristiken und der Abstimmung von Hyperparametern verringern. Erste Fortschritte beim Reinforcement Learning haben gezeigt, dass sie typische Optimierungsalgorithmen nullter und erster Ordnung lernen und bestehende Algorithmen übertreffen können, insbesondere in Bezug auf die Generalisierungsfähigkeit. Allerdings ist das Training immer noch sehr zeitaufwändig. Eine besondere Herausforderung ist das Training von Modellen auf Funktionen mit freien Parametern. Eine Änderung dieser Parameter (die beispielsweise Umweltfaktoren in einem realen Beispiel darstellen könnten) wirkt sich auf die zugrunde liegende Zielfunktion aus. Robuste Lösungen hängen daher vom gründlichen Sampeln ab, welches sich negativ auf den Zeitaufwand auswirkt. In dieser Arbeit haben wir das Laufzeit-Bottleneck des Reinforcement Learning Algorithmus identifiziert und waren in der Lage, die Laufzeit durch die Verteilung des Datensammelns drastisch zu reduzieren. Darüber hinaus haben wir die Auswirkung der Kombination von heterogenen Sampling-Strategien in Bezug auf die Generalisierungsfähigkeiten des gelernten Algorithmus untersucht.

Contents

List of Figures

List of Algorithms

Acronyms

DRL Deep Reinforcement Learning. 15

FMDP Finite Markov Decision Process. 19

GIL Global Interpreter Lock. 33

IC Integrated Circuit. 15

LTO Learn-to-optimize. 15

MDP Markov Decision Process. 17

MSE Mean Square Error. 26

MSLE Mean Square Logarithmic Error. 41

NN Neural Network. 20

PPO Proximal Policy Optimization. 16

PSV Post-Silicon-Verification. 45

RL Reinforcement Learning. 15

RNN Recurrent Neural Network. 15

TF Tensorflow. 29

TF-Agents Tensorflow Agents. 16

1 Introduction

Modelling the world became increasingly more mathematical. Thus, solving optimization problems became a tool in all major business divisions and fields of research like economics, finance, physics, biology, civil engineering, machine learning and many more. Due to the vast variety of conditions and circumstances in which these optimization problems need to be solved, many different algorithms and approaches have been developed to increase the performance and computation speed; however, this tends to be tedious, since the algorithm designer must study the problem and use his theoretical knowledge and empirical observations in many iterations to surpass the performance of existing solutions. This is especially sub-optimal when it comes to time-critical applications.

Another issue is, that most functions in the real world have no analytical expression, therefore zeroth order (derivative-free) or naive gradient optimization strategies are used to calculate point-wise solutions. These algorithms may be suitable for single evaluations. However, when considering an optimization problem of type $f(c_1, \ldots, c_k, x_1, \ldots, x_n)$ where f has to minimized multiple times dependent on free parameters c_i, established zeroth order algorithms have to be evaluated from scratch each time. We try to tackle this problem using a deep reinforcement learning strategy, following [DPRL21]. Given a set of free parameters, we're able to learn an optimization strategy which represents a likelihood distribution on where the minimum resides. Predicting a new optimum is simply to compute the arg max of our learned distribution. As mentioned, the runtime of successive evaluations improves drastically; however, the initial time expenditure of training a policy can be costly.

In this thesis, we analyse the bottleneck of time consumption in our training algorithm. By *distributing* the deep reinforcement learning process, we try to improve our existing solution in the following ways: Firstly, we aim to drastically decrease the runtime of training the model. Secondly, by distributing the data collection in our training process, we explore if using multiple sampling strategies simultaneously increases the generalization capabilities of the trained model. We evaluate this on three common benchmark functions, namely the sphere, ackley and rosenbrock function. We choose these functions, due to having a wide range of complexity and properties.

2 Related Work

In the last decade Deep Reinforcement Learning (DRL) was part of massive breakthroughs in the field of machine learning. 2013, Mnih et al [MKS+13] presented a deep reinforcement learning model which is capable of learning to play Atari games only using its raw pixels as input and surpassing expert level play in some of them. Several different research advancements made that possible. Most importantly: the use of neural networks replacing large state-action-tables to represent target functions. It reduced the need in feature engineering, expert level knowledge and simultaneously allowed to counteract the curse of dimensionality by significantly reducing needed storage and training data. Silver et al [SHS+17] showed a great demonstration of the capabilities of DRL by creating Alpha Zero, a program that self learned chess, go and shogi and outperformed at-the-time world-champion programs in each case.

Early works have described "Learning to learn" as the problem of deriving knowledge of related tasks to achieve better or faster learning results [HYC01; TP12]. Today, this is referred to as transfer learning and can be done by training a Recurrent Neural Network (RNN) which then later is fit to data using classical optimizers [ZL16].
Different work started stating the optimization algorithm search as a learning problem, called Learn-to-optimize (LTO).
Compared to Zoph et al [ZL16], Andrychowicz et al [ADC+16] directly parameterised the optimizer. This approach can be seen as using supervised learning to learn an optimization algorithm for supervised learning. It also produces a RNN. This time however, to fit *other* models to data.
Li et al [LM16] framed the learning task from a reinforcement learning perspective. They represent an optimization algorithm as a policy. Learning the algorithm was done by guided policy search and outperformed existing hand-engineered optimizers - notably in convergence speed and/or the final result.

Domanski et al [DPRL21] successfully used the reinforcement learning approach to train optimization algorithms in the Post-Silicon-Verification context. In order to increase performance of Integrated Circuits (ICs), variables and registers (called "tuning knobs") may be adjusted depending on given operating conditions. Reinforcement learning was used, to find a policy that maps conditions to tuning knobs. Existing state-of-the-art zeroth order optimization methods provide a tuning configuration for a given set of conditions. However, these solutions may only retain their quality until these conditions change, in which case the optimization has to be performed again. Reinforcement Learning (RL), which does not produce point wise solutions but a function, a so called tuning law, simply needs to reevaluate its learned policy. This greatly decreases runtime of subsequent evaluations at the cost of initially investing more time to train the RL model. The learned model could also be easily transferred to customer devices. That would allow ICs to constantly update their tuning configurations given conditions like battery life or temperature changes, without the need of heavy computational resources.

Mirhoseini et al [MGY+21] developed a chip floor planning algorithm that produces similar or superior performing net-lists compared to expert human designs, but in a fraction of the time. They posed the chip design problem as a reinforcement learning task, and trained an agent using Proximal Policy Optimization (PPO) to place macros on a chip. In [YSJ+22], Yue et al present a re-implementation of the `Circuit Training` framework, changing to the Tensorflow Agents (TF-Agents) Actor-Learner-API. They showed massive improvements in time to convergence, using both distributed data collection (scaling up to 256 workers) and distributed learning, using up to eight GPUs.

3 Theoretical Background

In this chapter, we will introduce the theoretical concepts behind learning optimization algorithms. This will include an overview over RL, the machine learning method we apply, as well as Markov Decision Processes (MDPs), the mathematical framework behind RL.

3.1 Learn-to-optimize

LTO has a long history. Initially, it was understood as speeding up training by transfer learning (re-using models from related tasks) or multi-task learning (learning related tasks together). Nowadays, LTO is described as the machine learning task to find an optimization algorithm which optimizes a specific or group of objective functions.

Li et al [LM16] presented a framework, depicted in Algorithm 3.1, that describes first order optimization algorithms. Common to all algorithms is, that they start with a sample s_0 (usually chosen randomly) that is improved iteratively. However, the way in which the sample is improved varies from algorithm to algorithm.

```
1  def optimize(objective_function f):
2      x_0 = random point in domain of f
3      i = 0
4      while True:
5          i += 1
6          Δx = π(f, {x_0, ..., x_{i-1}})
7          if stopping condition:
8              return x_{i-1}
9          x_i = x_{i-1} + Δx
```

Algorithm 3.1: Structure of first order optimization algorithms [LM16, Algorithm1]

Algorithms differ through choices of π. We present some examples:
Random search:

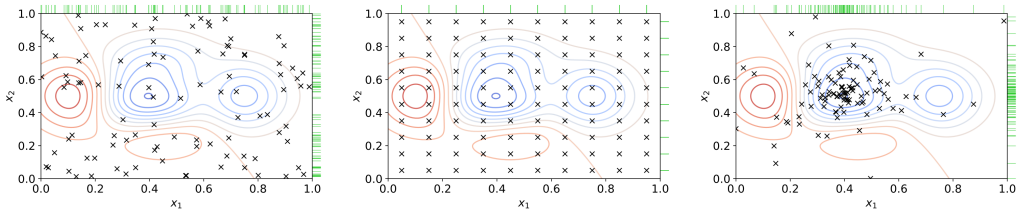
$$\pi(f, \{x_0, \ldots, x_{i-1}\}) = \left(\underset{x \in \{x_0, \ldots, x_{i-1}, x_r\}}{\arg\max} f(x) \right) - x_{i-1}$$

where x_r is another random point in the domain of f.

Gradient descent:

$$\pi(f, \{x_0, \ldots, x_{i-1}\}) = \gamma \, \nabla f(x_{i-1})$$

where γ denotes the learning rate and ∇f denotes the vector of partial derivatives of f.



(a) Random Search **(b)** Grid Search **(c)** Bayesian optimization

Figure 3.1: Sampling distributions [Com19]

Gradient descent with momentum:

$$\pi(f, \{x_0, \ldots, x_{i-1}\}) = \gamma \left(\sum_{j=0}^{i-1} \alpha^{i-j-1} \nabla f(x_{i-1}) \right)$$

where γ denotes the learning rate, ∇f denotes the vector of partial derivatives of f and α denotes the momentum of decay factor.

These examples show how optimization algorithms can be derived in this framework by choosing the right π. The question remains: how to choose a suitable π? The difference between procedures lies in the sampling distribution. In Figure 3.1, three different algorithms try to minimize an underlying two-dimensional function f. The green markers indicate the distribution of each parameter. Random Search and Grid Search have in common, that the sampling distribution does not correlate with the proximity to local or global minima. "Smart" algorithms like gradient descent or bayesian optimization try to exploit knowledge about gradients or previously sampled points to skew the sampling distribution towards more promising areas. An optimal policy π_* would be the deterministic distribution:

$$p(f, x) = \begin{cases} 1 & x \text{ is minimum of } f \\ 0 & \text{otherwise} \end{cases}$$

The goal in LTO therefore resides in developing a program that can compute π_* itself. Since executing any π is identical to executing a corresponding policy in a MDP (defined in Section 3.2.2), finding an optimal π_* can be formulated as the reinforcement learning task of maximising the reward function $r(s_t, a_t, s_{t+1}) = -f(s_{t+1})$.

3.2 Reinforcement Learning

Currently, a very popular strain of machine learning is supervised learning, where models are trained by providing labeled training examples. The model parameters are changed to minimize the calculated loss between the prediction of the model and the true label of the example. In RL, we learn a function, called *policy*, that maps situations to actions. This is different to supervised learning in the sense, that we do not provide any expert knowledge through labels. In contrast to unsupervised learning, RL algorithms do not try to find hidden structures in the training data but instead try to maximize a numerical reward signal. Typically in RL, the algorithm is responsible for

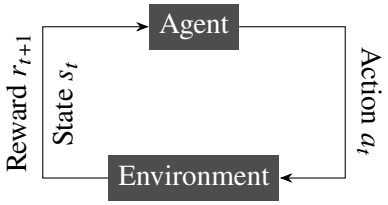


Figure 3.2: State - Action - Reward Loop between Agent and Environment

collecting its own training data. This mimics trial and error learning from humans and animals. This is done by interacting with an environment, which represents the problem that needs to be solved. It returns the observations and rewards in response to actions done by the RL agent.

In a series of discrete time steps[1] $t = 1, 2, 3, \ldots$ the agent gets a representation of a state s_t. He then takes an action a_t depending on his current policy π. The agent receives a reward r_{t+1} encoding the quality of following state s_{t+1}. This is called the reinforcement learning loop and is visualized in Figure 3.2.

A policy maps states to actions. The goal of the Reinforcement Learning Algorithm is to learn an optimal policy $\pi_* : \mathcal{S} \to \mathcal{A}$. A policy is an optimal policy π_*, if it maximises the total future reward for all states $s \in \mathcal{S}$. We formalize this using Markov Decision Processes. The policy as well as the transition function of the environment $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ might be a probability distribution.

3.2.1 Markov Decision Process

In RL, problems are typically stated as a MDP. It is defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$ where:

$\mathcal{S}$ is the set of all states,

$\mathcal{A}$ is the set of all actions,

T is the Transition model, where $T(s, a, s') = p(s'|s, a)$,

r is the reward function, where $r(s, a, s') \in \mathbb{R}$ and

γ is the discount factor, where $\gamma \in [0, 1]$.

If $\mathcal{S}, \mathcal{A}$ are finite sets, then T is a discrete probability distribution and we call it a Finite Markov Decision Process (FMDP).

MDPs are a mathematical framework used for modelling decision making in a stochastic process. RL uses MDPs where the probability distribution of the Transition model or the rewards is unknown. This builds on theoretical statements, e.g., the Bellman equations [Bel57]. By assuming that $T(s, a, s') = p(s'|s, a)$, we assume that the *Markov Property* holds, in other words: the next state s' depends only on the current state s and action a. This refers to the memoryless property of our stochastic process; however, it also implies, that all significant information for our agent is encoded in the current state s.

[1] For the sake of completeness: continuous-time markov decision processes also allow decision making at any time in trade-off for increased complexity in theory and application.

3.2.2 Learn-to-optimize as Markov Decision Process

Let $f : \mathbb{R}^n \to \mathbb{R}$ be the objective function. We aim to minimize f within the hypercube $[-b, b]^n$. We define our LTO Problem as a MDP as follows:

$$\begin{aligned} \mathcal{S} &= [-b, b]^n \\ \mathcal{A} &= [-b, b]^n \\ T(s, a, s') &= \begin{cases} 1 & s' = a, \text{ where } a \in \mathcal{A} \\ 0 & \text{otherwise} \end{cases} \\ r(s, s') &= f(s) - f(s') \\ \gamma &\in [0, 1) \,. \end{aligned} \tag{3.1}$$

When considering $\vec{x} \in [-b, b]^n$ as a (e.g single precision) floating point vector, we actually defined a FMDP by limiting $\mathcal{S}$ and $\mathcal{A}$ to finite sets. In theory, this FMDP could be solved analytically using the bellman equation [Bel57] or iteratively by policy iteration [How60]. However, the state space is too large to work in practice. Therefore we rely on Monte-Carlo Methods and more specifically Policy Gradient Methods, as they have proven to be successful in many applications [SLM+15; SWD+17].

Policy Gradient Methods do not learn some sort of value or action-value function to estimate the value of actions but instead use a parameterised policy that selects actions without consulting a value function[2] [p. 321 SB18]. For our tests we use a Policy Gradient Method called *Proximal Policy Optimization* [SWD+17]. This is done by using a policy π, where $\pi(a|s, \boldsymbol{\theta}) \in (0, 1)$, which is differentiable with respect to its parameter θ. In practice, the policy is represented by a Neural Network (NN). Since the specific algorithms are not of importance to this thesis, we leave it to the reader to further research this topic. A good introduction to Policy Gradient Methods and *REINFORCE* is given by Sutton and Barto [see p.321-331 SB18].

[2] The *REINFORCE Algorithm with baseline*, which we use for our runtime benchmarks, actually improves the basic *REINFORCE* algorithm by using an approximate state-value function [p.329 SB18].

3.2.3 Free parameters

In many domains, the same objective functions

$$f : \mathbb{R}^n \to \mathbb{R}, (x_1, \ldots, x_i, c_1, \ldots, c_k) \mapsto y$$

need to be minimized multiple times.
For example: $c \in \mathbb{R}^k$ may represent conditions of an environment like temperature or energy costs. If the conditions change, we need to re-compute the minimum of f. By simply changing the definition of the MDP as follows:

$$\begin{aligned} \mathcal{S} &= [-b, b]^n \\ \mathcal{A} &= [-b, b]^k, k \leq n \\ T(s, a, s') &= \begin{cases} 1 & s' = (a_1, \ldots, a_k, s_{k+1}, \ldots, s_n), \text{ where } a \in \mathcal{A}, \quad s, s' \in \mathcal{S} \\ 0 & \text{otherwise} \end{cases} \\ r(s, s') &= f(s) - f(s') \\ \gamma &\in [0, 1) \,. \end{aligned} \tag{3.2}$$

Here we limit the action space to all but the free parameter c_i. If we now find an optimal policy π_*, we find the minimum of f given conditions $c \in \mathbb{R}^{n-k}$, by evaluating $\pi_*(x_1, \ldots, x_k, c_1, \ldots, c_{n-k})$ for any $x \in \mathcal{A}$. This has the advantage that computing any minimum is a simple evaluation of π_*. This is much faster [DPRL21] and uses less computational resources than minimizing f again.

Note that optimization problems, where the target function has no free parameters, simply represent a subset of MDPs as in (3.2), where the optimal policy would be a constant function $\pi_*(s) = x_*$, where x_* is the global minimum of f.

3.2.4 Terminology

When we talk about reinforcement learning, we use terms like *agent* or *reward* as well as quantifiers like *episodes* and *steps*. To clarify their meaning, we construct a basic LTO example. A **training step** describes one complete loop in the RL training cycle: collecting data and learning from it (see: Figure 3.2). The data collecting process includes the interaction with the environment. One sequence of aligned time steps, starting with s_0 (starting state) and ending in s_T (terminal state), is called an **episode**[3]. The agent (or driver) will receive a starting state from the environment (black point in Figure 3.3), and perform actions depending on its current policy. The act of performing actions in a state is sometimes confusingly also referred to as (taking a) **step in the environment** or **time step**. Each time step consists of at least one tuple of state, action, reward and next state. Potentially additional information like a discount factor, or classification of states as starting or terminal state is included. In Figure 3.3, the red episode performs two steps, the blue episode three. Both episodes used the same policy to collect data. A collection of one or more episodes is called **trajectory**. Handing it over to the agent to learn and update its policy concludes a training step.

[3] Also sometimes referred to as "trial".

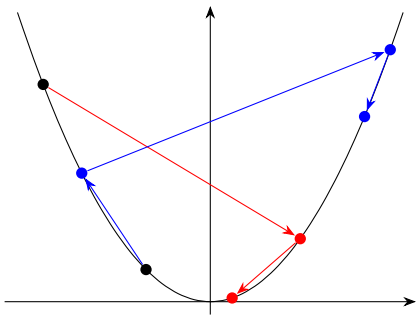


Figure 3.3: LTO example

3.2.5 Exploration vs Exploitation

In RL, every method faces the same dilemma. A policy, which tries to maximise all future rewards best to its current knowledge, will always choose the action, that is currently known to be optimal. However, in order to learn the *true* optimal policy, an agent must discover the value of all actions, which can only be done by following a sub-optimal policy. And since the action space is usually infeasibly large, the agent faces the question of which actions to prioritise.

In the context of sampling distributions, the agent has to choose between sampling near already promising results or sampling actions/points where the uncertainty is high. This dilemma is called the Exploration/Exploitation trade off. Different methods use different strategies to tackle this problem; however, all procedures have a stochastic element in common. The most basic idea, which proves to be very powerful, is called ϵ-greedy. The policy chooses a random action and therefore explores the action space with probability ϵ.

Another idea is to either randomize [MAF+16] or specifically set the starting state [YTL+23] of each episode. This is called *exploring starts* [p.96 SB18]. In relation to Equation (3.2), we make heavy use of this idea, by controlling the sampling strategy in which our starting states are computed. Besides choosing starting states randomly (*Random Sampling*), we also use *Latin hypercube sampling* [MBC79] and *Uncertainty Sampling* [LG94]. In Section 4.2.4 we explore, whether using different strategies simultaneously increases the ability of the agent to generalize.

4 Methodology

In this chapter, we state the problem definition, as well as describe the experiments we use, to evaluate our implementation.

4.1 Problem definition

The goal of this thesis is to lower the runtime of learning a policy. Therefore, we extend the reinforcement learning method as described in Sections 3.2.2 and 3.2.3 with a distribution strategy. The policy learned using this approach should perform as well as the policy learned in the sequential algorithm.

The second goal is to use distribution to run different RL methods in parallel. We want to evaluate whether combining training strategies in this way improves the generalization capabilities of our trained model. Specifically, we will run multiple different samplers in parallel, which are used to calculate the starting state s_0.

4.2 Experiment design

We evaluate our distribution strategy on multiple benchmark functions. The sphere, ackley and rosenbrock function cover a wide variety of properties and complexity. To make the implementation easier, we will normalize each function. We define runtime and performance tests and specify our tooling and our expectations.

4.2.1 Benchmark functions

Ackley

$$f(x,y) = -20\exp\left[-0.2\sqrt{0.5\left(x^2+y^2\right)}\right] - \exp\left[0.5\left(\cos\left(2\pi x\right)+\cos\left(2\pi y\right)\right)\right] + e + 20 \tag{4.1}$$

The Ackley function is a non-convex function which has its local minima at point $x = y = 0$. It has many local minima, which are sitting on a near flat plane in the outer regions. The minima is a big parabolic shaped hole in the origin of the function domain. The function is widely adopted as a

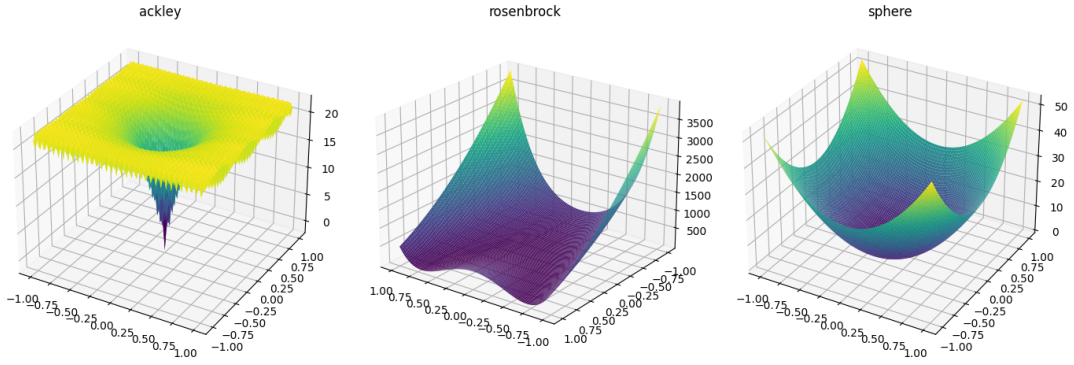


Figure 4.1: Benchmark functions

benchmark for optimization algorithms.
Domain: $x, y \in [-32.768, 32.768]$.
Global Minimum: $f(0,0) = 0$.

Rosenbrock

$$f(x,y) = \big(1-x\big)^2 + 100\big(y-x^2\big)^2 \tag{4.2}$$

The Rosenbrock Function is a valley shaped function. It is a non-convex function, with its minimum inside a long, narrow, parabolic shaped valley. Even though finding the valley is trivial, finding the global minimum is difficult.
Domain: $x, y \in [-2.048, 2.048]$.
Global Minimum: $f(1,1) = 0$.

Sphere

$$f(x,y) = x^2 + y^2 \tag{4.3}$$

The Sphere function is one of the most trivial optimization functions. It has only one global and no local minimum.
Domain: $x, y \in [-5.12, 5.12]$.
Global Minimum: $f(0,0) = 0$.

Normalization

For convenience reasons, we normalize each function:
Let $x_i \in [a,b]$ be the domain of benchmark function f. We normalise

$$\tilde{f}(x_1,\ldots,x_n) = f\left(\tilde{x}_1,\ldots,\tilde{x}_n\right), \text{ with } \tilde{x}_i = x_i \cdot \frac{b-a}{2} + \frac{b-3a}{2}.$$

Since our domains all have the form $[-a, a]$, we can simplify to:

$$\tilde{f}(x_1, \ldots, x_n) = f\left(\tilde{x}_1, \ldots, \tilde{x}_n\right), \text{ with } \tilde{x}_i = x_i \cdot \frac{b-a}{2}.$$

In all cases the new domain of $\tilde{f}$ is now $x_i \in [-1, 1]$ and the new global minimum, if the old one was at x^*, is now at $\tilde{x}$:

$$\tilde{x}_i = x_i^* \cdot \frac{2}{b-a}.$$

Figure 4.1 shows a 3D visualization of the normalised benchmark functions.

4.2.2 Runtime tests

To get a good grasp of the runtime performance of our implementation, we evaluate it for multiple degrees of distribution. As we describe in Chapter 5, we differentiate between data collection and learning. We define data collection, as all actions taken by the driver to collect training data and operations needed to create one single trajectory. As training, we define all actions taken by the agent, to update its policy given a trajectory. Actions needed to broadcast the updated policy to the driver, are measured separately.

For measuring time, we will use pythons `time` module. In the distributed case, we measure as follows:

```
import time

t_collecting = 0
t_training = 0
t_broadcasting = 0

for step in 1,..., training_steps:
    t_start = time.time()
    # broadcasting
    t_broadcasting += time.time() - t_start

    t_start = time.time()
    # data collecting
    t_collecting += time.time() - t_start

    t_start = time.time()
    # training
    t_training += time.time() - t_start
```

Algorithm 4.1: Runtime distributed case

In the sequential case, broadcasting is unnecessary, since policy information of driver and agent is stored in the same memory object.

As for the hyperparameters we choose the values in Table 4.1.

[1] Total number of episodes collected.
[2] Number of drivers that run in parallel.

Function:	Ackley, Rosenbrock and Sphere
Dimensions	2
Training steps:	10.000
Time steps:	2
Episodes[1]:	2^6
Drivers[2]:	$2^0, \ldots, 2^6$
Sampling:	Random Sampling

Table 4.1: Hyperparameter Runtime Tests

We run all tests using a 24 core Intel Skylake and a Tesla P100.

We expect runtime of t.raining to stay mostly consistent across degrees of distribution. The runtime of data collection however, should drop significantly. Due to the fact, that the problem is basically *embarrassingly parallel*, we expect to roughly halve the runtime of data collection by doubling the amount of collection workers.

4.2.3 Performance Comparison: Sequential vs Distributed

The main goal of this suite of experiments is to verify, that our distributed algorithm produces indeed identical results to the sequential algorithm. We do this by observing whether the training process behaves in a similar fashion. Because of the stochastic nature of RL, we compute multiple training runs and choose the one producing the best policy.

Function:	Ackley and Rosenbrock
Dimensions	2
Training steps:	5.000
Time steps:	2
Episodes:	30
Drivers:	1 and 3
Sampling:	Random Sampling

Table 4.2: Hyperparameter Distributed Sampling Test

To evaluate the quality of a policy, we compute the Mean Square Error (MSE) between the prediction of the model and some ground truth after every 10 training steps.

Because the model does not predict the global minimum, but instead a minimum given free parameters $\{c_1, \ldots, c_k\}$, we perform multiple evaluations. In the two dimensional case, in which we have one free parameter c, we decided to evaluate $f(x, c)$ for all $c \in \{-1, -0.9, -0.8, \ldots, 1\}$. To compute a ground truth estimate in respect to a fixed free variable, we use Monte-Carlo-Optimization.

Every ten training steps during training the model, we will compute the models prediction $f(x, c)$ for all $c \in \{-1, -0.9, -0.8, \ldots, 1\}$ and the MSE to our estimated ground truth. We then compare the sequential and distributed training progress. As hyperparameters, we choose the values described in Table 4.2.

We expect the convergence properties of sequential and distributed learning to be identical, i.e. over multiple evaluations, it should be impossible to distinguish between distributed and sequentially learned models.

4.2.4 Generalization Capabilities of Heterogeneous Sampling

We want to extend the idea of *exploring starts* by using a heterogeneous sampling strategy as defined in Section 5.3.4. To observe potential differences, we run and measure the performance of each sampling strategy using homogeneous sampling in the same way as described in Section 4.2.3. Namely we compare random (5.3.1), latin-hypercube (5.3.2) and uncertainty sampling (5.3.3). To benchmark heterogeneous sampling, we use three drivers using each sampler once. The specific hyperparameters used are defined in Table 4.3.

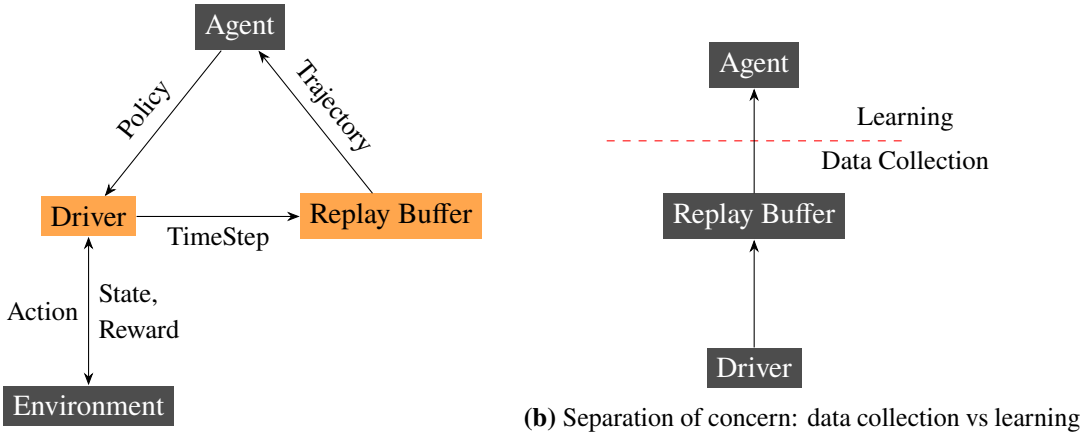
Function:	Ackley, Rosenbrock
Dimensions	2
Training steps:	5.000
Time steps:	2
Episodes:	30
Drivers:	3
Sampling:	Random Sampling, Latin Hypercube Sampling, Uncertainty Sampling and Heterogeneous Sampling

Table 4.3: Hyperparameter Heterogeneous Sampling Test

5 Implementation

5.1 TF-Agents

Our distributed data collection algorithm builds on top of Tensorflow[1] and TF-Agents[2]. Tensorflow (TF) is a free and open source library from Google. Its main alternative is PyTorch [3], which is developed by the Linux Foundation. Even though there may be small differences for some expert systems or research domains, both libraries can be used successfully to develop complex machine learning and reinforcement learning applications. However, since we extend an existing TF implementation to solve LTO-problems using TF-Agents, we continue using it to research runtime benefits of distributed calculations in RL.



(a) The RL-Loop in TF-Agents

(b) Separation of concern: data collection vs learning

Figure 5.1: Reinforcement Learning with Tensorflow-Agents

TF-Agents is a TF extension and is a robust, fast and scalable RL library. It assists in creating state-of-the-art RL models by providing a vast variety of algorithms, abstractions to build environments, as well as an Actor-Learner API to support asynchronous distributed reinforcement learning across multiple CPUs and accelerators. TF-Agents extends the basic RL loop (Figure 3.2) with useful components like Replaybuffers and Drivers. The typical RL training loop is portrayed in Figure 5.1a.

[1] https://www.tensorflow.org/
[2] https://www.tensorflow.org/agents
[3] https://pytorch.org

Instead of the agent, we now use a *driver* to interact with the environment. The idea is to separate the concern of data collection from learning. An example implementation of Figure 5.1a is portrayed Algorithm 5.1.

```
for step in 1,..., training_steps:
    # update driver policy
    # In the sequential case, this is done implicitly.
    # The driver and agent use the same policy object in memory.
    driver.policy.update(agent.policy)

    # run driver and write collected data into replay buffer
    driver.run()

    # get all collected data
    experience = replay_buffer.gather_all()

    # train the agent, update the agents policy
    agent.train(experience)
```

Algorithm 5.1: RL training loop

In the first step, we run the driver. The driver will interact with the environment defined in the initialization in the following way:

```
def run(self):
    for episode in 1,..., number_of_episodes:
        # reset the environment
        current_state = environment.reset()

        for step in 1,..., steps_per_episode:
            # use action defined by current collect_policy
            action = collect_policy.get_action(current_state)

            # take step in environment
            reward, next_state = environment.take_action(action)

            # append time_step to replay buffer
            replay_buffer.append((current_state, action, reward, next_state))

            # set new current state
            current_state = next_state
```

Algorithm 5.2: Pseudo code of driver interacting with environment

An episode resembles a sequence of state, action, reward, next state (s_t, a_t, r_t, s_{t+1}) tuples. In different episodes the driver may start from different starting states s_0; however, the policy used for data collection stays identical. When the `run` method has completed, the driver will have written the collected trajectories[4] into the replay buffer. This is then given to the agent, which will use this data to improve its policy.

[4] We call a sequence of aligned time steps *trajectory*

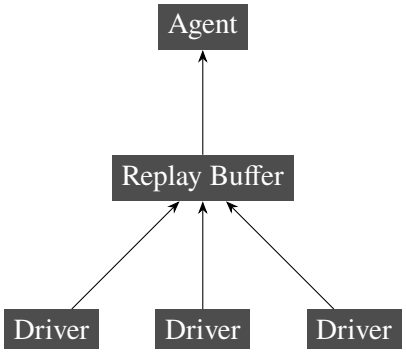


Figure 5.2: Parallel data collection

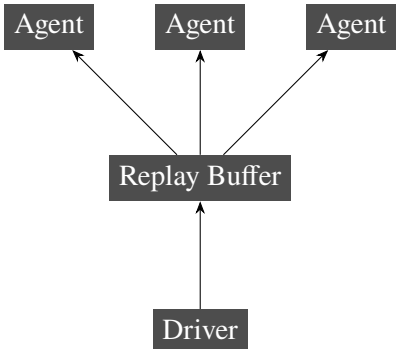


Figure 5.3: Parallel learning

5.2 Parallel data collection vs. learning

As a reminder: our main goal is to reduce the runtime of training our reinforcement learning model without compromising the quality of the predictive quality of our policy. More specifically, we are looking for a distribution strategy, which allows vertical scaling when training the model. When thinking about distributing the reinforcement learning process, it is helpful to separate the data collection process and the learning procedure as denoted in Figure 5.1b. This results in the question, what to actually parallelize: Data collection, learning or both?

The parallelization of data collection is depicted in Figure 5.2. We run multiple drivers in parallel, then merge the collected data into the same replay buffer. The big advantage is, if the merging is done right, the learner cannot distinguish between two episodes collected by a single driver and one episode collected by two drivers respectively. While collecting, the drivers do not need any synchronisation. The problem is conceptually embarrassingly parallel.

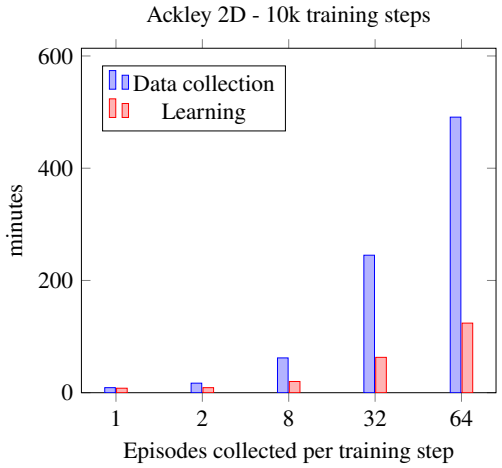


Figure 5.4: Time consumption: Data collection vs training depending on number of episodes collected

Parallelization of the training can be more challenging, depended on the RL algorithm used for training. That said, TF-Agents provides distribution strategies to enable computations across multiple GPUs. The learner "receives a batch of training data, splits it across the GPUs, computes the forward step, aggregates and computes the mean of the loss, computes the backward step, and performs a gradient update" [YSJ+22]. This is a similar approach as done in supervised learning. The key difference is, that training data needs to be collected in the first place. Parallelization of training (and reducing training time) therefore increases the data collection effort. Figure 5.4 shows, that especially for larger numbers of episodes collected per training step, the data collection clearly is the bottleneck, even when not performing distributed learning.

The number of collected episodes (let's call it batch size for now) can also have a similar effect as in supervised learning, where by increasing the number of samples, we better approximate the "true" gradient of the loss. By increasing the batch size, updates should become less noisy; however, finding the right amount of episodes can be a challenging hyperparameter optimization problem and should be approached with care. In a nutshell: distributed data collection is a necessity for high performing, distributed RL systems.

5.3 Sampling

In Section 3.2, we described the RL loop. An agent chooses an action a_t depending on the current state s_t. The environment returns a new state s_{t+1} based on the transition model $\mathcal{T}$ as well as an reward r_{t+1}. We have not discussed, how the environment chooses s_0.

In certain types of problems, choosing s_0 may seem straightforward. It may represent the starting state of a game or the initial sensor inputs of a robot. Section 3.2.5 touched the topic of exploring starts. Our agent cannot explore the whole state space by choosing actions. Due to the free parameters we have $\mathcal{A} \subsetneq \mathcal{S}$. The environment needs to ensure a high variety of different starting states; otherwise, the agent will not be able to generalize over the whole function domain. We use three different *sampling strategies* (samplers for short), to tackle this problem.

5.3.1 Random Sampling

Random sampling chooses s_0 by using a random function $\text{rand}(l, r)$, that returns a value $x_i \in [l, r]$ with equal probability. So for $\mathcal{S} = [-1, 1]^n$:

$$s_0 = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \text{ where } x_i = \text{rand}(-1, 1) \tag{5.1}$$

It's easy, fast and unbiased, making it a great choice for basic benchmarks.

5.3.2 Latin Hypercube Sampling

Latin hypercube sampling extends random sampling by dividing the interval $(-1, 1)$ of each variable into M equally probable intervals.

$$s_0 = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \text{ where } x_i = \text{rand}(m_l, m_r) \tag{5.2}$$

where (m_l, m_r) represents the interval, where the least points are already sampled from. If multiple intervals have equally few samples, a random interval is chosen. This is done for each x_i. The idea is to enforce a representative variability, and increase stability when training the agent.

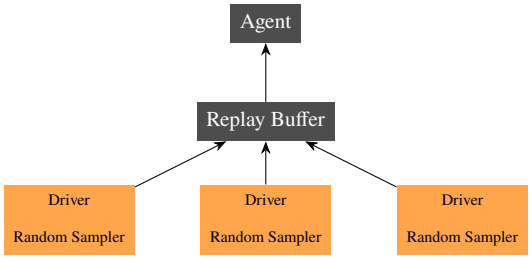


Figure 5.5: Distributed Homogeneous Sampling

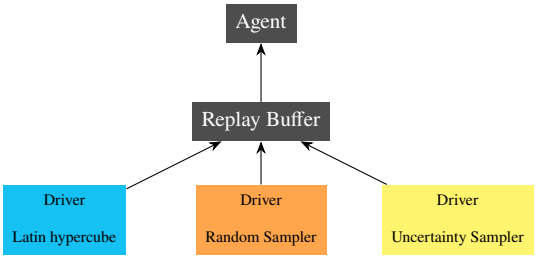


Figure 5.6: Distributed Heterogeneous Sampling

5.3.3 Uncertainty Sampling

In Section 3.2.5 we discussed the challenges of exploring the state space. When thinking about learning conceptually, the most progress can be achieved in areas where we lack the most knowledge. We try to encode this in a sampling strategy, where we explore areas of the state space, i.e the function domain, with which our agent is least familiar with. For this, we take a large number of random samples p_i, and evaluate the entropy of the distribution of $\pi(p_i)$. The Entropy $H(X)$ encodes the average level of "information" of a random variable X.

$$H(X) = -\sum_{x \in X} p(x) \log p(x) = \mathbb{E}[-\log p(X)]$$

The higher the entropy, the higher is the uncertainty of the agent. Therefore we return the sample p_i, where the entropy of the distribution of $\pi(p_i)$ is the highest.

$$s_0 = p_i, \quad \text{where } p_i = \underset{p_i \in \text{Random Samples}}{\arg\max} \; H\big(\pi(p_i)\big) \tag{5.3}$$

5.3.4 Heterogeneous Sampling

We define *homogeneous sampling*, as using one sampling strategy, e.g. random sampling exclusively. In contrast, assigning different sampling strategies to different drivers collecting data in parallel is called *heterogeneous sampling*. Figure 5.5 shows a version of homogeneous sampling, using three distributed random sampling drivers. Figure 5.6 shows three drivers, each using a different sampler.

5.4 Details

5.4.1 Multi-processing vs Multi-threading

In python, there are two common ways to use parallelization: multi-threading and multi-processing. Due to the Global Interpreter Lock (GIL), python does not allow running more than one thread at once. This makes multi-threading a valid choice only for handling multiple I/O-bound tasks "in

parallel". For performance-orientated tasks however, only multi-processing is a viable option. The multi-processing library spawns sub processes instead of threads. This side-steps the GIL; however, sharing data can be challenging. Python multi-processing proposes two ways of sharing data:

1. Shared Memory
2. Manager/Queues (communication via serialization)

Shared Memory is a useful tool for simple small values, arrays or ctypes. It's fast and easy to use, when writing is exclusive to a single process. **Managers, Queues or Pipes** can be used for arbitrary python objects - if they are serializable - which is not always given. Tensorflow objects often tend to fall into the category *not serializable*, which tributes to enormous difficulties managing shared memory between processes. The serialization is also more time consuming than just accessing shared memory. We will evaluate the impact of inter-process communication when analysing the runtime.

5.4.2 First approach

In Section 5.2, we discussed, that data collection is the runtime bottleneck. We distribute data collection by running multiple drivers in parallel. The first approach is to use `multiprocessing.Pool`, which offers a convenient way to parallelize execution of a function with different input values across processes. A main thread initializes all the components. Pythons `Pool`-function then initializes a set of processes. Each training step, the pool distributes the data (a pair of driver and replaybuffer) on each process and executes the `runner`-function.

```
1  def runner(driver: TFDriver, replay_buffer):
2      driver.run()
3      trajectory = replay_buffer.gather_all()
4      return trajectory
5
6  def learn():
7      # initialize drivers, replaybuffers, environment and agent
8      with Pool(number_of_driver) as pool:
9          for step in 1,..., training_steps:
10             trajectories = pool.map(runner, [(driver1, replaybuffer1), (driver2, replaybuffer2)
                   ...])
11             experience = merge_trajectories(trajectories)
12             agent.train(experience)
```

Algorithm 5.3: Distribution using Pythons multiprocessing.Pool

However, distributing the driver to each process failed, because it is not serializable. Therefore the initialization has to be done by each process itself.

5.4.3 Final implementation

Because each process needs to keep its own copy of a driver object, each process also needs to keep its own policy. After each training step, the policy in the main process is updated by the agent. However, since a policy itself is also not serializable, it can't be sent by default via pipes or queues.

To solve this issue, we implemented a serialization method which disassembles the policy into its trainable variables θ. These can be broadcasted to each process, where the policy then gets reassembled. For communication between processes, we decided to use `pipes`:

```
def learn():
    # initialize processes, agent, ...
    for step in 1,..., training_steps:
        for each process p:
            pipe_p.send(agent.policy.θ)

        trajectories = []
        for each process p:
            trajectories.append(pipe_p.recv)

        experience =
             merge_trajectories(trajectories)

        agent.train(experience)
```

Algorithm 5.4: Main process

```
def driver_loop(pipe, information for
     initialization)

    # initialization
    replay_buffer = ...
    environment = ...
    driver = ...

    while(True):
        # receive new policy from agent
        θ = pipe.recv()
        new_policy = assemble_policy(θ)

        # update driver policy
        driver.policy.update(new_policy)

        # run driver
        driver.run()
        trajectory =
             replay_buffer.gather_all()

        # send back trajectory
        pipe.send(trajectory)
```

Algorithm 5.5: Worker process

5.5 Limitations

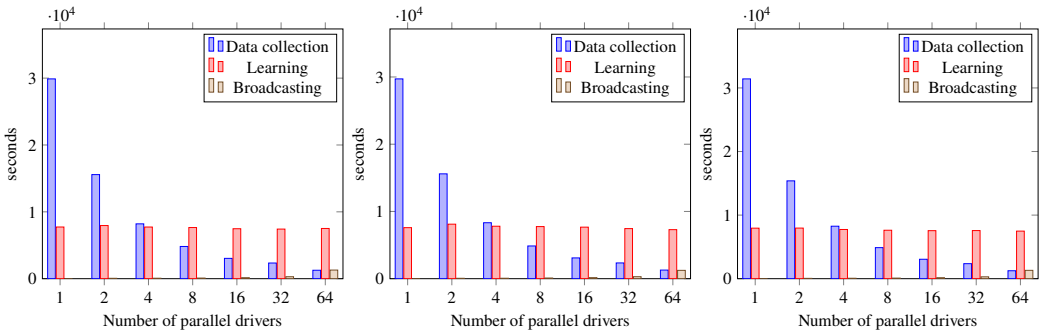
Firstly, distributing the data collection process as proposed in Section 5.2 has a clear drawback. The degree of distribution is limited by the number of episodes that are collected per training step. If the training algorithm or domain in which the RL algorithm is trained, require only one (or very few) episode(s) per training step, this distribution strategy fails.

Secondly, some extensions to this algorithm, e.g. using a grid-sampler, may require additional synchronisation between processes. Our approach assumes each worker process to be independent of the others.

6 Evaluation

We share the evaluations of the experiments defined in Section 4.2. Where our expectations are not met, we discuss possible reasons and sources of error.

6.1 Runtime



(a) Sphere 2D **(b)** Ackley 2D **(c)** Rosenbrock 2D

Figure 6.1: Runtime comparison

We compared runtime on three different benchmark functions as described in Section 4.2.2 in different degrees of distribution. The results are depicted in Figure 6.1. The x-Axis describes the number of drivers that ran in parallel. $k = 1$ therefore represents the sequential case. The blue bars represent the time spent on data collection, red bars represent the time spent on learning and yellow bars the time spent on broadcasting - all as defined in Algorithm 4.1.

On each function, we measured very similar results. This was to be expected, since all function evaluations are given through an analytical expression. In regards of the collection time, we predicted a time reduction multiplier of $\frac{1}{k}$ by using a distribution degree of k. We computed the mean over all 3 plots. The computation for one driver in the mean took $t^{(1)}_{collecting} = 30341s$. In Table 6.1 the different degrees of distribution we evaluated:

$$\rho = \frac{t^{(1)}_{collecting}}{t^{(k)}_{collecting}}$$

For a low degree of distribution, the performance is close to our target improvement of k. However, for higher degrees, we see a hard drop off in performance increase. We assume this happens due to the following reasons:

Number of Drivers k	Time collecting $t_{col}^{(k)}$	ρ	ρ^*
1	30341	1.0	1.0
2	15516	1.96	1.95
4	8259	3.67	3.64
8	4856	6.25	6.13
16	3062	9.91	9.44
32	2349	12.92	11.48
64	1264	24.01	11.93

Table 6.1: Runtime improvement multiplier

Firstly, the amount of episodes collected per driver are getting very small. For $k = 64$ we only collect a single episode per driver. This means, that comparatively the overhead of updating the policy in each driver becomes more and more significant.

Secondly, we observe a very poor runtime decrease from 16 to 32 drivers. Since we ran all tests on a 24 core CPU, only 24 drivers can collect episodes simultaneously. Therefore all other drivers are idle. But how can we explain a near perfect halving of runtime from 32 to 64 drivers? A closer look at Figure 6.1 displays a sharp increase in the time needed for broadcasting (yellow bar). We measured the time needed for the main process, to distribute the policy variables to all drivers. The prediction was, that the inter-process communication would be slow and time consuming. This turned out to be false. In Algorithm 5.4 we observe, that sharing the policy to all drivers is done via a `pipe.send()`. This is a blocking function, until `pipe.recv()` is called. However, when the CPU is working under full load, the main process blocks until each driver has received its policy. Depending on the scheduler of the operating system, that could result in 40 drivers finishing their computation, before the last driver can start its computation. Since we measure this time as time used for *broadcasting*, we only measure the actual runtime of the remaining 24 or less drivers.
When adding time used for collection and time used for broadcasting, and use this to calculate the adjusted improvement multiplier

$$\rho^* = \frac{t_{collecting}^{(1)}}{t_{collecting}^{(k)} + t_{broadcasting}^{(k)}}$$

we find that from $k = 32$ to $k = 64$ basically no performance increase can be measured. That makes sense, since only 24 drivers can run in parallel anyway. Figure 6.2 displays a visualization of ρ^* in respect to the theoretical maximum. The red line represents the hardware limit of 24 cores. We achieved half of the theoretical improvement factor. Collecting more episodes in total could increase the improvement factor even more. However, in our experience, collecting more than 32 episodes did not increase the training performance of the RL agent.

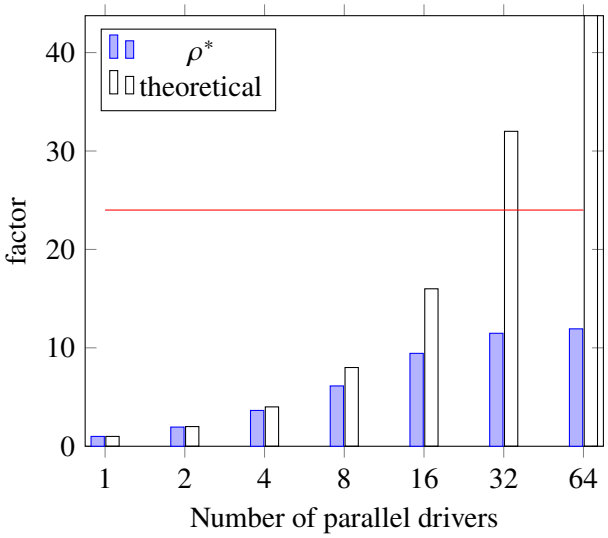
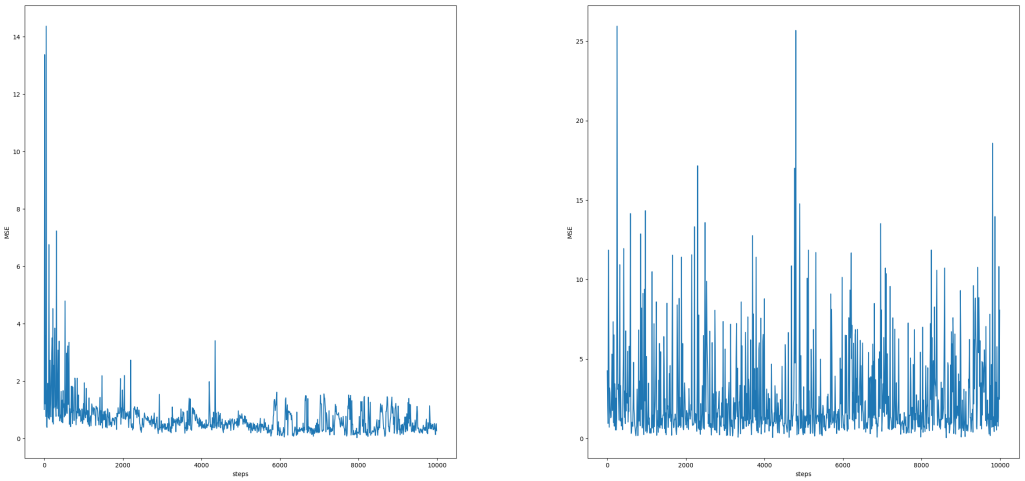


Figure 6.2: Runtime improvement multiplier

6.2 Performance homogeneous sampling

To compare the learning progress between sequential and distributed training, we collected the same number of total episodes per training step. We measured the MSE to the estimated ground truth. We expected two similar training progressions. As clearly visible in Figure 6.3, the plots *are* distinguishable. To be more precise, the distributed algorithm does not seem to converge or train at all.

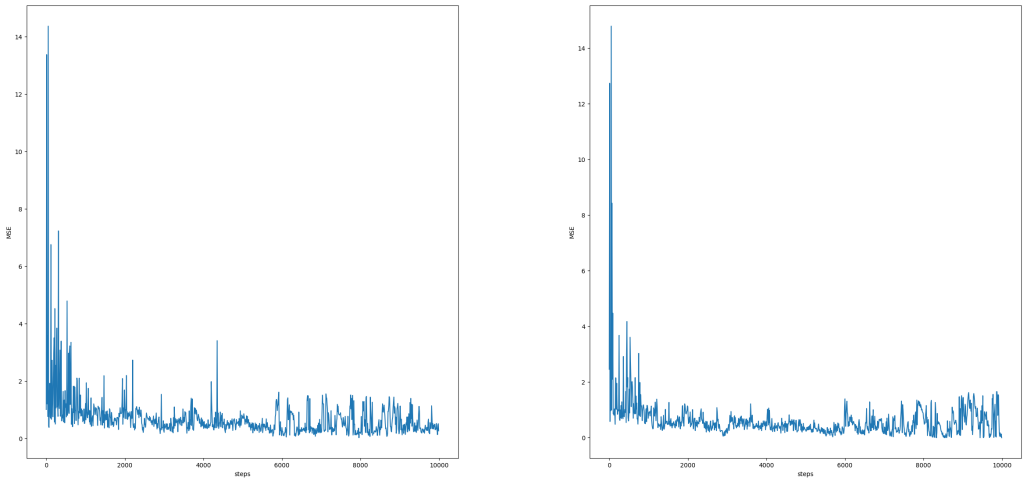


(a) Sequential Data Collection

(b) Distributed Data Collection

Figure 6.3: Sequential vs. Distributed Data Collection

We were not able to identify the exact reason where this error results from. However, to investigate whether our method of running multiple drivers in parallel makes sense, we build a pseudo-distributed algorithm. We basically kept everything the same, but instead of multi-processing, we used multi-threading. We still initialized multiple drivers and merged the different small trajectories into a single big one. We made a deepcopy of the policy object for each driver[1], and updated them using the same method as in the distributed case.



(a) Sequential Data Collection **(b)** Pseudo-distributed Data Collection

Figure 6.4: Sequential vs. Pseudo-distributed Data Collection

Figure 6.4 shows the agent using the pseudo-distributed data collector in comparison with the sequential one. Figure 6.4a and Figure 6.4b are very similar, as we originally predicted. This verifies, that merging multiple trajectories works as intended. It also suggests that updating the policy with our handmade serialization method works. Because our algorithm only fails, when using true multi-processing, we expect the driver or sampler to be needing additional data, which is not being updated in his private memory. These complication could indicate, that using c++ with multi-threading would have been a better choice than manually distributing data collection in python. Another option would have been to implement the Actor-Learner-API. However, this is mainly used for asynchronous collecting and learning, which might makes results hard to compare.

This also indicates, that the runtimes discussed in Section 6.1 have to be taken with a grain of salt. Fixing the performance problem in the distributed learner may effect the overall runtime. However, we still are convinced, that distributing data collection is the right approach to decrease runtime when using RL for LTO.

[1] In multiprocessing, all processes have different memory and own copies of the different objects. In the sequential case, the driver and agent share the same policy object. To simulate the distributed case, we copied the policy so that each driver had its own policy object in memory. We updated them, using the same method as described in Section 5.4.3.

6.3 Performance heterogeneous sampling

Using the pseudo-distributed algorithm, we computed benchmarks using each of the different samplers described in Section 5.3. We ran each training three times, and used the example with the best policy in any training step. Figures 6.5a to 6.5c display the training process of the homogeneous samplers. Figure 6.5d depicts the training process of the agent, which used one driver with a different sampler each.

Table 6.2 shows the variance of each training in the first and last 1000 training steps. The agent using the novel, heterogeneous sampling showed a significantly faster convergence, than the homogeneous approaches - which is reflected in the variance. However, in later stages the homogeneous sampling strategies performed better overall.

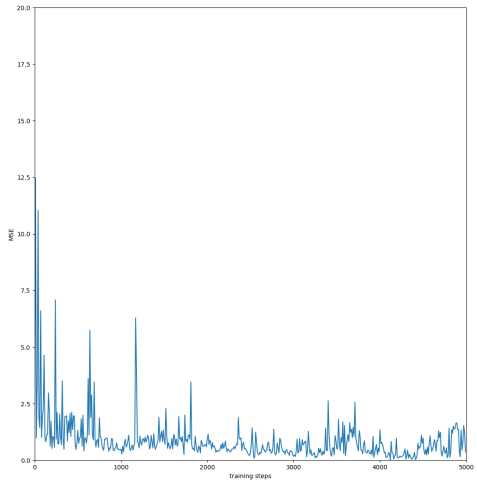
Sampler	First 1000 steps		Last 1000 steps		Overall
	Mean	Variance	Mean	Variance	Best
Random	1.6	3.6	0.6	0.2	0.018
Latin hypercube	1.9	4.47	0.4	0.03	0.00081
Uncertainty	2.3	2.92	0.4	0.09	0.04
Heterogeneous	1.3	1.59	0.6	0.07	0.15

Table 6.2: Convergence properties of different sampling strategies on 2D Ackley function

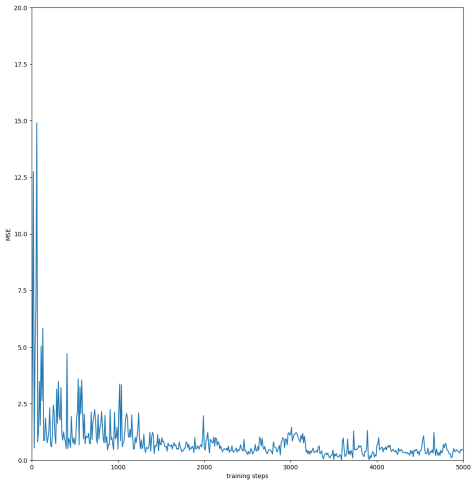
We repeated the same tests on the rosenbrock function, this time with 10000 training steps. Due to the very large function values for the free parameters $c_i < 0$, we use the Mean Square Logarithmic Error (MSLE) instead of the MSE. As visible in Figure 6.6, all agents struggled to converge on this function. Still, the agent using heterogeneous sampling (Figure 6.6d) clearly outperformed all homogeneous sampling strategies (Figures 6.6a to 6.6c). In Table 6.3, we can see that on the last 1000 training steps, the agent using heterogeneous sampling averaged an MSLE of $\frac{1}{3}$ lower than the agents using homogeneous sampling.

Sampler	First 1000 steps		Last 1000 steps		Overall
	Mean	Variance	Mean	Variance	Best
Random	17.1	71.8	15.3	19.5	3.0
Latin hypercube	19.4	45.1	16.1	11.7	2.8
Uncertainty	16	48.9	14.5	36.2	1.1
Heterogeneous	15.1	62.2	9.7	14.0	1.4

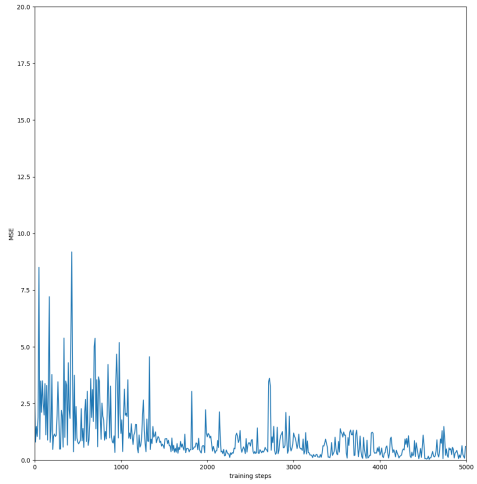
Table 6.3: Convergence properties of different sampling strategies on 2D Rosenbrock function



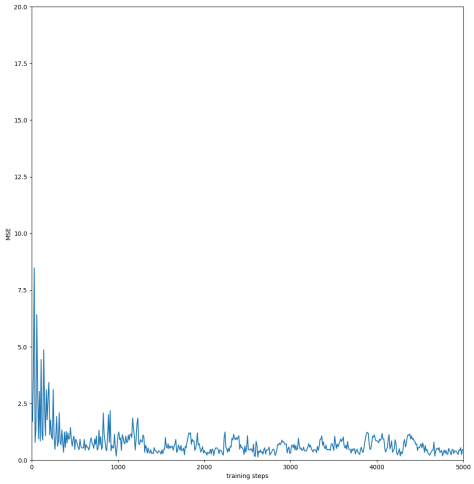
(a) Random Sampler



(b) Latin Hypercube Sampler

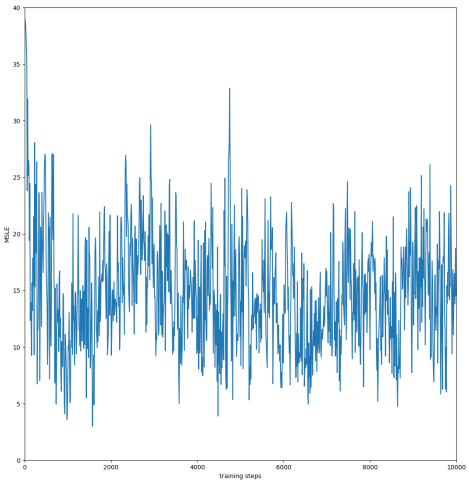


(c) Uncertainty Sampler

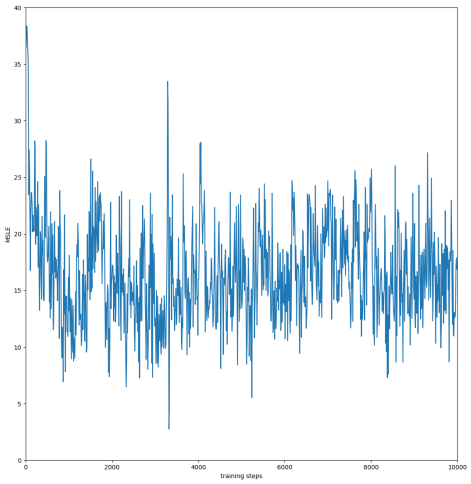


(d) Heterogeneous Sampler

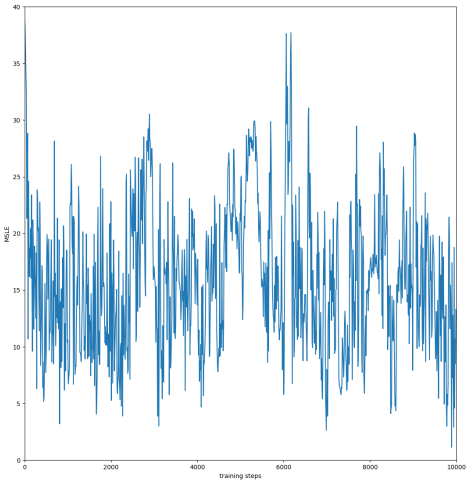
Figure 6.5: Heterogeneous Sampling vs Homogeneous Sampling on 2D Ackley function



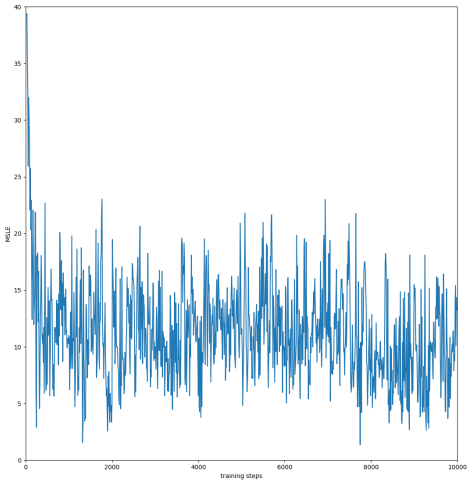
(a) Random Sampling



(b) Latin Hypercube Sampling



(c) Uncertainty Sampling



(d) Heterogeneous Sampling

Figure 6.6: Heterogeneous Sampling vs Homogeneous Sampling on 2D Rosenbrock function

7 Conclusion

This thesis discusses a strategy to distribute Deep RL in the LTO context. We provide insight into the advantages and challenges of distributing data collection using Python and the open source frameworks TF and TF-Agents.

We show, that distributed data collection can decrease runtime significantly, to the point where training exceeds data collection in time consumption. This enables algorithms to collect larger quantities of data in parallel, which decreases the time to convergence in many applications. It provides the foundation to further reduce runtime in LTO applications through parallel training, because distributed learning strategies rely on higher volumes of training data.

Future work may collect and train asynchronously as done by [MGY+21; YSJ+22]. This has the potential to speed up the training process even more. It has to be shown, whether the performance of the model would remain equally good. This could be done using the Actor-Learner-API and then evaluated for usability and performance.

Furthermore, we provide insight in using the abstraction of distributed data collection to include novel strategies like heterogeneous sampling to further increase performance of RL models. However, future work can extend this idea to sampling on multiple environments simultaneously. E.g. in Post-Silicon-Verification (PSV), where multiple prototypes are built and tested, a surrogate of each device might represent an environment. Therefore training on multiple environments could support computing a tuning law, that generalizes well over all devices.

Lastly, distributing data collection should not only decrease runtime in the context of LTO. Any RL task where collecting large amounts of data helps the agent learn the optimal policy π_* faster, could reduce runtime by distributing data collection. Policy gradient methods might profit in particular.

Bibliography

[ADC+16] M. Andrychowicz, M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul, N. de Freitas. "Learning to learn by gradient descent by gradient descent". In: *CoRR* abs/1606.04474 (2016). arXiv: 1606.04474. URL: http://arxiv.org/abs/1606.04474 (cit. on p. 15).

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957 (cit. on pp. 19, 20).

[Com19] W. Commons. *Hyperparameter optimization*. 2019. URL: https://en.wikipedia.org/wiki/Hyperparameter_optimization (cit. on p. 18).

[DPRL21] P. Domanski, D. Pflüger, J. Rivoir, R. Latty. "Self-Learning Tuning for Post-Silicon Validation". In: (Nov. 2021). arXiv: 2111.08995 [cs.LG] (cit. on pp. 13, 15, 21).

[How60] R. A. Howard. *Dynamic Programming and Markov Processes*. The MIT Press, 1960 (cit. on p. 20).

[HYC01] S. Hochreiter, A. S. Younger, P. R. Conwell. "Learning to Learn Using Gradient Descent". In: *Artificial Neural Networks — ICANN 2001*. Ed. by G. Dorffner, H. Bischof, K. Hornik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–94. ISBN: 978-3-540-44668-2 (cit. on p. 15).

[LG94] D. D. Lewis, W. A. Gale. "A Sequential Algorithm for Training Text Classifiers". In: *CoRR* abs/cmp-lg/9407020 (1994). arXiv: cmp-lg/9407020. URL: http://arxiv.org/abs/cmp-lg/9407020 (cit. on p. 22).

[LM16] K. Li, J. Malik. *Learning to Optimize*. 2016. DOI: 10.48550/ARXIV.1606.01885. URL: https://arxiv.org/abs/1606.01885 (cit. on pp. 15, 17).

[MAF+16] W. Montgomery, A. Ajay, C. Finn, P. Abbeel, S. Levine. "Reset-Free Guided Policy Search: Efficient Deep Reinforcement Learning with Stochastic Initial States". In: *CoRR* abs/1610.01112 (2016). arXiv: 1610.01112. URL: http://arxiv.org/abs/1610.01112 (cit. on p. 22).

[MBC79] M. D. McKay, R. J. Beckman, W. J. Conover. "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code". In: *Technometrics* 21.2 (1979), pp. 239–245. ISSN: 00401706. URL: http://www.jstor.org/stable/1268522 (visited on 05/20/2023) (cit. on p. 22).

[MGY+21] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, J. Dean. "A graph placement methodology for fast chip design". In: *nature* 594.7862 (June 2021), pp. 207–212. DOI: 10.1038/s41586-021-03544-w. URL: https://ui.adsabs.harvard.edu/abs/2021Natur.594..207M (cit. on pp. 16, 45).

[MKS+13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller. “Playing Atari with Deep Reinforcement Learning”. In: (Dec. 2013). arXiv: 1312.5602 [cs.LG] (cit. on p. 15).

[SB18] R. S. Sutton, A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on pp. 20, 22).

[SHS+17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815 (cit. on p. 15).

[SLM+15] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, P. Abbeel. *Trust Region Policy Optimization*. 2015. arXiv: 1502.05477 [cs.LG] (cit. on p. 20).

[SWD+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG] (cit. on p. 20).

[TP12] S. Thrun, L. Pratt. *Learning to learn*. Springer Science & Business Media, 2012 (cit. on p. 15).

[YSJ+22] S. Yue, E. M. Songhori, J. W. Jiang, T. Boyd, A. Goldie, A. Mirhoseini, S. Guadarrama. “Scalability and Generalization of Circuit Training for Chip Floorplanning”. In: *Proceedings of the 2022 International Symposium on Physical Design*. Ed. by L. Behjat, S. Yang. ISPD ’22. Association for Computing Machinery, 2022, pp. 65–70. DOI: 10.1145/3505170.3511478. URL: https://doi.org/10.1145/3505170.3511478 (cit. on pp. 16, 31, 45).

[YTL+23] D. Yin, S. Thiagarajan, N. Lazic, N. Rajaraman, B. Hao, C. Szepesvari. *Sample Efficient Deep Reinforcement Learning via Local Planning*. 2023. arXiv: 2301.12579 [cs.LG] (cit. on p. 22).

[ZL16] B. Zoph, Q. V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *CoRR* abs/1611.01578 (2016). arXiv: 1611.01578. URL: http://arxiv.org/abs/1611.01578 (cit. on p. 15).

All links were last followed on May 29, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature