Saarland University

Bachelor Thesis

Mapping Instructions to Ports: A Reinforcement Curriculum Learning Approach

Submitted by:

Luis Paulus

Submitted on:

January 6, 2021

Reviewers:

Prof. Dr. Verena Wolf

Prof. Dr. Sebastian Hack

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis. I marked the parts of this thesis where content or phrasing has been taken from others. For drawings, sketches or schemes as well as for graphical depictions I have stated whenever they were created by someone else.

I declare that the written and electronic versions of this thesis are fully identical.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Luis Paulus
Saarbrücken, January 6, 2021

Abstract

The job of a processor is to execute incoming instruction sequences. This process has been optimized over time and is therefore quite complex for modern processors. Modern processors have several ports to execute instructions and thus can make use of instruction-level parallelism to speed up execution time. This is the reason why the port mapping, which maps the instructions to the ports, is an essential component of such processors. Unfortunately, manufacturers normally do not publish these port mappings for their processors. Nevertheless, knowing these mappings would help to optimize programs.
In this thesis, we apply reinforcement learning, in particular a combination of tabular q-learning and curriculum learning, to infer a processor's port mapping. SMT solvers are a well-known approach to solve optimization problems. We compare ourselves to an SMT solver, which mostly produces optimal results for smaller mappings, but currently cannot handle larger mappings. Our approach produces good results for small and large mappings. In contrast to the SMT solver, it is scalable to experiments with many instructions, even though it was only trained on the former, equally producing good solutions. Although we limit ourselves to mappings that do not consider the decomposition of instructions into μops, we are the first to address the problem with reinforcement learning. Our results, especially the scalability, are promising and raise the wish to extend these studies in future work.

Acknowledgments

I would like to thank my supervisor Prof. Wolf and also Prof. Hack for giving this interesting topic to me. Also, I would like to thank both for reviewing this thesis. I would like to thank my advisor Timo for his advices, I highly appreciate his support during the process of writing this thesis. Also, I would like to thank my advisor Fabian for his detailed answers to my questions. I would like to thank Jeremias and Alina for proofreading this thesis. Lastly, I would like to thank my family. My parents support me with everything I do, I am very grateful for that.

Contents

1 Introduction **1**

2 Processors & Port Mappings **3**
2.1 Data Hazards . . . 3
2.2 Processor Design . . . 3
2.3 Port Mapping . . . 5
2.4 Underlying Model . . . 5
2.5 Bottleneck Simulation Algorithm . . . 7
2.6 SMT Solver . . . 8

3 Related Work **9**

4 Reinforcement Learning **11**
4.1 Agent-environment-interaction . . . 11
4.2 Policy . . . 12
4.3 Return . . . 12
4.4 Value-functions . . . 13
4.5 Q-learning with Q-tables . . . 14
4.6 Optimistic Initialization . . . 15
4.7 Multi-agent Q-learning . . . 15
4.8 Curriculum Learning . . . 16

5 Applying Q-learning Methods to the Problem **19**
5.1 Construction of the Environment . . . 19
5.1.1 Creating the Secret Mapping . . . 19
5.1.2 Creating Experiments . . . 20
5.1.3 Defining the Error Function . . . 20
5.2 The Task . . . 20
5.3 The MDP . . . 21
5.4 Construction of the Algorithm . . . 22
5.4.1 Defining the Q-table . . . 23
5.4.2 Instructions as Agents . . . 24
5.4.3 Updating the Q-table . . . 25
5.5 Determining the Mapping Shape in Advance . . . 26
5.6 Defining the Training Curriculum . . . 28

6 Improving and Analyzing the Algorithms **31**
6.1 Increasing Solution Stability . . . 31
6.2 Mapping Shape in Advance . . . 34
6.3 Hyperparameter Search . . . 38
6.3.1 Number of Training Levels & Experiment Length . . . 38

6.3.2 Number of Experiments per Episode 41
6.3.3 Discount Factor 42
6.3.4 Experiment Distribution 45
6.4 Influence of the Mapping 46

7 Evaluation **49**
7.1 Evaluating with Experiments of Length 2 51
7.2 Evaluating with Experiments of Length 5 54
7.3 Evaluating with Experiments of Length 20 56
7.4 Evaluating with Experiments of Length 100 58

8 Conclusion and Future Work **61**

1 Introduction

Modern processors can apply out-of-order execution to process instruction sequences [27]. This is a technique that allows the processor to freely rearrange incoming instructions as long as it does not affect the outcome. Using this technique leads to an intransparent execution process that is very difficult to understand. The manufacturers of processors normally do not publish which instructions can be executed on which ports. Understanding this process in its entirety helps to optimize programs as these programs can then make use of the process' characteristics (for example bottlenecks).
There exist different approaches to infer this internal information of a processor's port mapping, for example by using an evolutionary algorithm [24] or by applying machine learning [18].
In this thesis reinforcement learning strategies are applied to the problem of inferring a processor's port mapping. In particular, the developed algorithm makes use of curriculum learning [4, 20] and q-learning [25]. Q-learning is an algorithm that trains an agent in an environment to find the actions that lead to the best outcome. Curriculum learning describes the intuitive human learning process: starting with small problems and increasing the difficulty from time to time. By combining curriculum learning with q-learning the resulting algorithm is able to find good solutions for large port mappings (meaning port mappings for many instructions and many ports). The results of the algorithm are compared to the results of an SMT solver developed by Fabian Ritter. The solutions of the SMT solver cannot compete with the solutions of the reinforcement learning algorithm.
The theoretic model used in this work is developed by Ritter and Hack [24]. They designed an algorithm that predicts the time a processor needs to execute a given instruction sequence. This algorithm is used as a crucial part of the environment to train the reinforcement learning agents.
The thesis is structured as following. First, the fundamental theories of processors and port mappings are presented in chapter 2. Chapter 3 covers existing work related to this topic. Next, the background information is completed with chapter 4 about reinforcement learning. In chapter 5, reinforcement learning strategies are applied to the problem and the algorithm is designed. This algorithm is then run and analyzed in chapter 6. Chapter 7 concentrates on evaluating the algorithm with mappings of large size and comparing the results to those of the SMT solver. In the last chapter the findings are summed up and future work is discussed.

2 Processors & Port Mappings

2.1 Data Hazards

Modern processors are built in a way that allows the processor to execute instructions in parallel. While this reduces the required overall time to process instruction sequences, new problems arise with this technique: data hazards [29]. Data hazards describe the problem that occurs when instructions need to wait for the completion of a previous instruction. There are three types of data hazards:

- WAW: a write-after-write data hazard occurs when a write-instruction's destination register is a previous write-instruction's destination register and the execution of this previous instruction is not completed yet
- RAW: a read-after-write data hazard occurs when an instruction tries to read from a register to which a previously executed but not completed instruction writes
- WAR: a write-after-read data hazard occurs when an instruction writes to a register which a previously executed but not completed instruction reads from

2.2 Processor Design

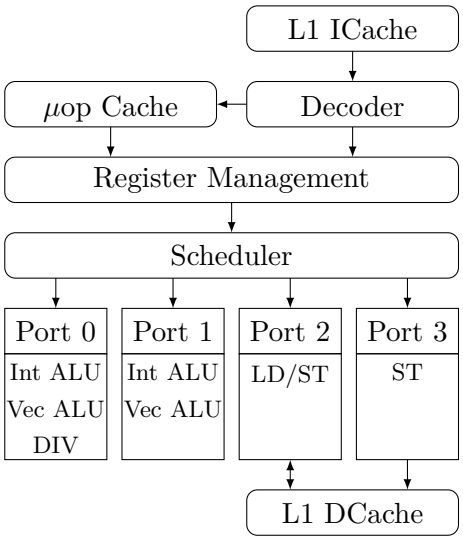


Figure 2.1: A simplified subdivision of a modern processor (taken from [24])

The task of a processor is to process incoming instruction sequences. A simplified subdivision of a modern processor can be seen in Figure 2.1. The processor has to process incoming instruction sequences which are stored in the L1 Instruction Cache. Every instruction can be decoded in its so-called micro-operations (μops). The job of the decoder is to decode these incoming instructions to the corresponding μops. Because the μops will probably be needed again in the future they will also be stored in a cache. Modern processors can change the execution order of incoming instructions as long as that does not change the outcome. To be more specific, the processor has to make sure that the read-after-write dependencies between all operations are preserved, but as long as this regulation is observed it can rearrange the program and execute instructions in parallel. This technique is called out-of-order execution [27]. The idea behind this strategy is that every cycle an instruction can be executed. That would not necessarily be the case if the processor had to execute them in the incoming order because then the processor would have to wait for each instruction to be completed to execute the next one. So by deploying this strategy the processor can reduce the overall execution time significantly.
Since read-after-write dependencies are observed, remaining potential data hazards are violations against write-after-write and write-after-read dependencies. Resolving these is the task of the register management.
Next, the μops have to be executed on the appropriate execution units. Some execution units exist more than once such that parallel execution of similar μops is possible. Each processor has a defined number of ports behind which the execution units are grouped. The execution units in modern processors are often pipelined so that each port can start executing a new instruction (or more specific new μops) each cycle. As an example, it can be seen that in Figure 2.1 port 0 has an integer arithmetic logic unit (ALU), a vector ALU and a divide unit and that port 0 and port 1 have an Int ALU. So some μops have several ports as an execution option. Therefore there is a scheduler that decides for each incoming μop on which port it will be executed.
The required time a processor needs to execute an incoming instruction sequence depends on the dependencies of the instructions and the port mapping. In this work, we only consider instruction sequences that do not have dependencies. That is the reason why we concentrate on which instructions are mapped to which ports in a given processor. Further, we assume that the instructions are scheduled optimally and that each instruction does block a port for exactly one cycle. As described in [24] these two assumptions normally hold for modern processors.

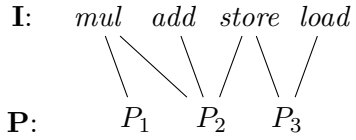


Figure 2.2: A port mapping in the two-level model

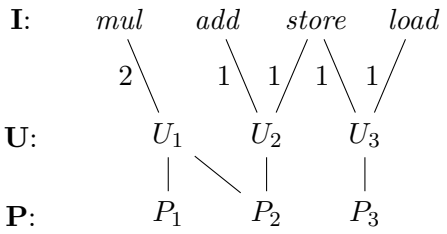


Figure 2.3: A port mapping in the three-level model

2.3 Port Mapping

A port mapping is a model that describes which instruction is decomposed into which μops and which μops can be executed on which ports for a given processor. An example can be seen in Figure 2.3: first, there are the instructions *mul*, *add*, *store* and *load*, the second level consists of the μops U_1-U_3, and the third level consists of the ports P_1-P_3. Because of these three levels, we call such a mapping a *mapping in the three-level model* [24]. For this thesis, it is sufficient to further abstract from this three-level model and consider a two-level model only, which is defined as follows.

Definition 1. A *port mapping in the two-level model* is a bipartite graph $(\mathsf{I} \dot\cup \mathsf{P}, M)$ with the nodes split disjointly into a set I of instructions and a set P of ports and edges $M \subseteq \mathsf{I} \times \mathsf{P}$ between these [24].

This means that a mapping is a graph that defines which instruction can be executed on which port. So we do not consider the decomposition of instructions into their μops and instead simply map each instruction to its possible execution options. Each instruction has to have at least one port as its execution option. Normally there do exist instructions that can be processed without a functional unit. For example, a move instruction that will only lead to a change in the register mapping. But in this work, we only consider instruction sequences where each instruction can be mapped to at least one port.
An example of a mapping in the two-level model can be seen in Figure 2.2. Compared to the mapping in the three-level setting in Figure 2.3, it can be seen that the two-level model comes with some restrictions in the representation. It is not possible in the two-level model to indicate that an instruction might be decomposed in more than one μop. Furthermore, there can exist diagonal connections in the three-level model that might miss in the two-level model.

2.4 Underlying Model

The throughput is a measure that represents how long a certain processor needs to execute a given program (or sequence of instructions). There do exist processors (e.g. those by Intel) that have an integrated performance counter which counts the number of μops that were executed for each port. Such counters can be used to infer port mappings. But we do not want to rely on this feature as not all processors have it and accordingly that would restrict the applicability of the results of this work. That is why we stick with the throughput as the measure of the processor's execution effort for given instruction sequences.

Definition 2. The *throughput* $t^*(e)$ of an instruction sequence (or experiment) e on a given processor is the average number of processor cycles required to execute e in a steady state [24].

We say that an execution reached a steady state if during an infinitely long execution the average cycle number per iteration stays the same from some point on.
As already stated, a processor's port mapping strongly affects the number of cycles that are needed to execute an incoming instruction sequence. To understand this process we have to take a look at the following definition of throughput depending on a given mapping as defined in [24].

Definition 3. Given a port mapping $m := (\mathsf{I} \dot\cup \mathsf{P}, M)$ in the two-level setting, the *throughput* $t^*_m(e)$ under m for an experiment $e : \mathsf{I} \to \mathbb{N}$ is the objective value of an optimal solution to the following linear program:

$$
\begin{array}{lll}
\text{minimize} & t & \\
\text{subject to} & \displaystyle\sum_{k \in \mathsf{P}} x_{ik} = e(i) & \text{for all } i \in \mathsf{I} \qquad \text{(A)} \\
& \displaystyle\sum_{i \in \mathsf{I}} x_{ik} \le t & \text{for all } k \in \mathsf{P} \qquad \text{(B)} \\
& x_{ik} \ge 0 & \text{for all } (i,k) \in M \qquad \text{(C)} \\
& x_{ik} = 0 & \text{for all } (i,k) \notin M \qquad \text{(D)}
\end{array}
$$

An experiment is defined as a function that maps an instruction to a natural number. It represents for every instruction the number of its occurrences. The order of these instructions is negligible since we only consider instruction sequences that do not have dependencies. $x_{ik} \in \mathbb{R}$ describes how much of the mass of instruction i is distributed to port k. Constraint (A) describes that for a fixed i the distributed mass of this instruction to each port has to sum up to its total mass. So it simply means that each instruction's mass can be distributed to different ports. Constraint (B) states that the throughput t is the upper bound for each port's accumulated mass. As port k is fixed in the formula the sum of all the instruction mass allocated on this port is less or equal than t. So t is the greatest accumulated mass that one of the ports has. The fact that the mass of the instructions is only distributed to ports where they are allowed to be executed on is ensured by the last two constraints: constraint (C) ensures that each instruction's allocated mass on a port is greater or equal to 0 as long as the mapping m allows executing i on k; constraint (D) ensures that if there does not exist an edge in the mapping m, the mass of i cannot be allocated on port k.
It might seem counterintuitive that x_{ik} is a real number as it is not possible to split up an instruction. But the intuition behind this linear program becomes clearer when taking a look at an example. Let $e := \{add \rightarrowtail 1, mul \rightarrowtail 2, load \rightarrowtail 1\}$ be the given

experiment and the processor use the port mapping from Figure 2.2. Then it can be seen in Figure 2.4 how each instruction's mass is distributed to the three ports. The left axis shows the required cycles. For each port, there is a bucket where the corresponding instruction mass is put. The throughput is 1.5 as that is the upper bound for all buckets. The allocation of *add* and *load* is straightforward because they can only be executed on the corresponding ports according to the mapping. There remain two times *mul*. The first *mul* is allocated to P_1 since that bucket is still empty and allocating it to P_2 would lead to a throughput of 2.0. That would conflict with the minimization constraints in the linear program. For the second *mul* it can be seen that it is split among P_1 and P_2. This is possible for the processor by executing the second *mul* in every second iteration on P_1 and in every other second iteration on P_2. So just looking at one of those iterations the upper bound would be 2.0 because of either P_1 or P_2. But the throughput is the average number of required processor cycles per execution in a steady state, so the resulting throughput is 1.5. That is why the allocation mass of an instruction to a port is not an integer but a real number, it does not mean that the instruction is actually split.

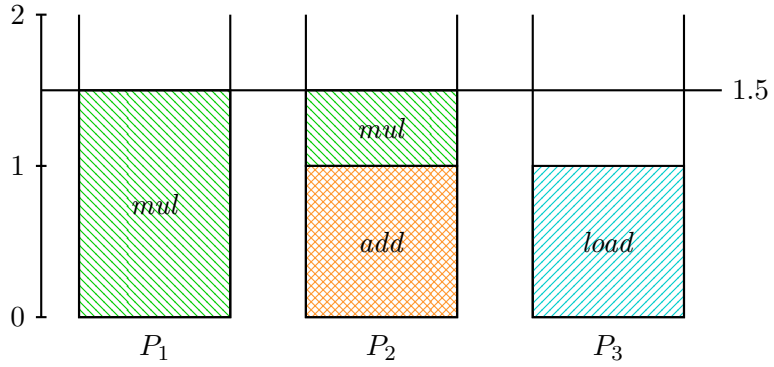


Figure 2.4: An example for port usage (similar to the one in [24])

The linear program as defined in Definition 2 can also be defined for the three-level model, but we will not introduce it here. We explained the three-level setting to give an insight into the operational sequence of a processor and to demonstrate how processors use port mappings. But the algorithm we will introduce is only constructed for mappings in the two-level model. This is the reason why we will only consider two-level mappings from now on, and instead of calling it a two-level mapping we simply stick with *(port) mapping*.

2.5 Bottleneck Simulation Algorithm

The bottleneck simulation algorithm developed by Ritter and Hack [24] is used to simulate the throughput of a given experiment. As solving the linear program from

Definition 3 for each experiment would take too much time, instead

$$t^*_m(e) = \max_{Q \subseteq \mathsf{P}} \frac{\sum\{e(i) \mid Ports(m,i) \subseteq Q\}}{|Q|} \tag{1}$$

is solved to determine the throughput $t^*_m(e)$ of an experiment e, given a port mapping $m := (\mathsf{I} \dot\cup \mathsf{P}, M)$. Here $Ports(m,i) := \{k \mid (i,k) \in M\}$, so it describes the set of ports under mapping m where instruction i can be executed on. The idea of Equation (1) is to find the ports that are the bottleneck, meaning the ports that have the highest throughput for the given experiment. The fraction stands for the throughput of a given subset of ports as the numerator represents the number of instructions in the experiment that can only be executed on these ports and the denominator represents the size of this port subset. By taking the maximum with regards to any subset of ports Equation (1) represents the optimal throughput of the experiment as long as the instructions are scheduled to the ports optimally. Otherwise, the throughput $t^*_m(e)$ of the given experiment will be higher than the result of the algorithm. So the result is a lower bound for the throughput. It is proven that the result of this algorithm is equivalent to the solution of the linear program in the publication by Ritter and Hack [24].

2.6 SMT Solver

The problem of determining if a given formula is satisfiable is called *satisfiability* [8]. A formula consists of quantifiers, variables, logical connectives, and function and predicate symbols. If an interpretation of the variable and these symbols can be found so that the result is *true*, the formula is called satisfiable. To interpret these symbols, so-called *background theories* can be given as constraint to make sure that certain symbols are interpreted the correct way. This is the so-called *satisfiability modulo theory* (SMT).
Fabian Ritter developed an algorithm that makes use of an SMT solver to find a port mapping. It generates experiments up to predefined length and tries to find a mapping that can explain the generated throughputs using an SMT solver. The predefined parameter is called *instruction bound*. So by increasing this parameter the resulting mapping is more stable for experiments of greater length. If it is set to a low value the resulting mapping produces equal throughputs up to that length, but for larger experiments the results may differ. Increasing the instruction bound leads to an exponential boost of additional possible experiments the algorithm has to check. In the evaluation chapter this algorithm is used for an additional comparison of the results. It is then referred to as the SMT solver, even though it only uses one.

3 Related Work

Finding out details and information about microarchitectures has already been the subject of research in the past. Agner Fog published instruction tables [10] for many x86 microarchitectures by Intel, AMD, and VIA. The tables contain for all instructions their latency, throughput, and micro-operation breakdowns. Some CPUs have integrated hardware performance counters that can be used to measure the number of processor cycles an instruction sequence needs to be executed. These are used by Agner Fog in combination with manually constructed microbenchmarks to obtain the information about the throughput and the subdivision into micro-operations. Such counters are not integrated into every CPU. The instruction tables also contain data for some processors without them. To get information about port usage for instructions on such processors, the created instruction sequences are a combination of instructions with unknown port usage and those where other sources already published port usage properties. This approach is very cumbersome as creating these instruction sequences takes a lot of effort. But the resulting tables were the only source for these data for some time in the past. Abel and Reineke show in their work uops.info [1] that the port usage is underapproximated in these tables. In their approach, specific instructions are used to block a defined subset of the available ports so that the instruction that is analyzed is executed on a different port if possible. This principle is used for all instructions, leading to a more accurate outcome with regards to the port usage. These microbenchmarks are generated automatically. However, as their approach also makes use of hardware performance counters only port mappings of corresponding processors are inferred.
Google developed EXEgesis [11], a tool that can infer latencies, throughputs and the scheduling of μops for given instruction sequences for microarchitectures by Intel. They use documents provided by Intel containing this information. As these documents were initially addressed to human beings, the process of automatically parsing them complicates this approach. Additionally to these documents not being parsable well, they often miss information. llvm-exegesis [7] is a second project by Google inferring this missing information by evaluating experiments that are designed for this particular case. As the name tells, this second project is part of the LLVM-framework [15]. Nonetheless, this approach also makes use of integrated performance counters for these experiments to obtain the missing information.
There also exists a closed source tool by Intel, named Intel Architecture Code Analyzer (IACA) [12], that can model the throughput and micro-operation-to-port distribution given an instruction sequence. Although having insights in the processors that are not publicly available, IACA's predictions can still be different from observed outcomes as described by Abel and Reineke [1]. IACA is not being developed any further since April 2019.
Laukeman et al. developed OSACA [16], a tool for throughput analysis of loop kernels. The idea of OSACA is that it is an open-source version of IACA. They use existing port mappings and validate them using generated instruction sequences. Applying this tool

to processors that do not have an x86 architecture would require a port mapping to validate.
llvm-mca [5] is a tool that predicts the performance of code. It is also part of the LLVM-framework [15]. For its predictions the tool makes use of scheduling models provided by the framework.
There also exists a tool that makes use of machine learning, called Ithemal by Charith Mendis et al. [18]. It learns to predict the throughput for a given processor and is portable among different processor architectures. As usual with neural networks, the resulting model is a black box. Determining specific bottlenecks of a processor's port mapping is not possible with this tool due to the lack of access to the details of the prediction process.
Ritter and Hack developed PMEvo [24], a framework to infer a processor's port mapping. PMEvo does not depend on hardware performance counters. This is achieved by a freshly developed evolutionary algorithm that tries to find a mapping that can explain the throughput of given instruction sequences. These sequences are constructed in such a way that the port usage of the corresponding instructions can be inferred. Being independent of hardware performance counters makes this approach applicable to a much wider range of microarchitectures. An important part of the framework is a new bottleneck simulation algorithm that can simulate the throughput of experiments for a given port mapping efficiently. When using PMEvo to infer port mappings, the quality of the results is comparable to that of existing work.
Alex Renda et al. recently developed DiffTune [23], a tool that can infer microarchitectural parameters such as the number of micro-operations per instruction. It trains a differentiable surrogate function whose output for a set of parameters matches that of llvm-mca [5]. Then it trains parameters for the surrogate that make its predictions match the measurements. Using this tool, all parameters for x86 microarchitectures in llvm-mca [5] can be inferred.

Reinforcement learning is a paradigm of machine learning to train an agent to learn sequential decision making [25]. While applying reinforcement learning algorithms is originally designed for a setting with recurring agent-environment-interaction [25], it can also be applied to combinatorial optimization problems [17]. For example, Khalil et al. use reinforcement learning to solve classical graph problems such as the maximum cut, the minimum vertex cover or the traveling salesman problem [13]. In particular, they make use of q-learning [25], which is also used by others to solve combinatorial optimization problems [2, 6].

The problem of finding a processor's port mapping can also be seen as a combinatorial optimization problem. In this thesis, q-learning with q-tables is applied to the problem of inferring an unknown port mapping of a processor given throughputs of instruction sequences for this processor. The approach is to combine tabular q-learning with curriculum learning [4] to guide the learning path to find the unknown mapping. For the simulation of experiments, the simulation algorithm by Ritter and Hack [24] is used.

4 Reinforcement Learning

Reinforcement learning is the third machine learning paradigm besides the two well known *supervised learning* and *unsupervised learning* [25]. The difference between those two and reinforcement learning is that the latter describes the paradigm to solve a problem not by recognizing (un)labelled data and categorizing them but by trying to maximize given rewards [25]. So it is much more convenient to apply to a problem where the task is to learn sequential decision making in a given environment.

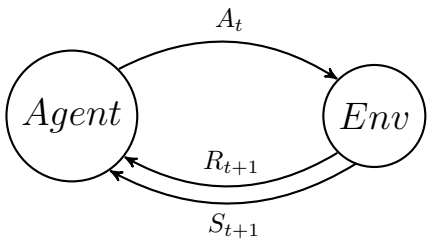


Figure 4.1: Agent-environment-interaction

4.1 Agent-environment-interaction

In reinforcement learning an agent is trained to choose actions that give high rewards [25]. The agent always interacts with a defined environment. That means that the agent which finds itself in state S_t at timestep t can act with an action A_t and at time step $t+1$ the environment will provide R_{t+1} to the agent. R_{t+1} is the reward corresponding to the preceding action A_t. This reward tells the agent how to evaluate the chosen action. In addition to the reward the environment also tells the agent the next state S_{t+1} it finds itself in after acting as chosen. This process can be seen in Figure 4.1. In this way a *trajectory* [25]

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \ldots$$

can be formed to describe the interaction between the agent and the environment. The environment can be seen as a Markov Decision Process which is defined as follows.

Definition 4. A *Markov decision process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ in which $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ a finite set of actions, $\mathcal{T}$ a transition function defined as $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and $\mathcal{R}$ a reward function defined as $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ [22].

$\mathcal{T}(s, a, s')$ describes the probability of ending in state s' when choosing action a in

state s. Choosing action a in state s leading to state s' will give a reward described by $\mathcal{R}(s, a, s')$. In a Markov decision process (MDP) it is not necessary that every action can be chosen in every state, the set of all possible actions in state s is therefore defined by $\mathcal{A}(s)$. However in this work, we will only work with MDPs where every action can be chosen in every state, so $\forall s \in S : \mathcal{A}(s) = \mathcal{A}$.

4.2 Policy

The agent tries to find the best action for any given state, meaning an action that results in high accumulated rewards. The goal is to find a policy that contains this information so that by simply following this policy the agent chooses actions that lead to high accumulated rewards.

Definition 5. An *action policy* is a function $\pi : \mathcal{S} \to \mathcal{A}$ that maps states to actions in such a way that future rewards a maximized [26].

Finding a policy that tells the agent which choices give the highest reward (short or long term) is not easy because the agent is faced with a problem: on the one hand the agent wants to follow the policy as it depicts which of the possible actions is the best, but on the other hand the agent has to choose other actions than the ones recommended by the policy as otherwise, it might never find out if there are other actions better in different states. So for the agent, it is a tradeoff between exploration and exploitation. Exploring means to choose different actions than suggested by the policy and exploiting means to strictly follow the policy for maximal rewards. This difficulty is often referred to as the *exploration-exploitation dilemma* [25, 28]. The agent has to do both to find a good policy. Only exploiting leads to not finding the optimal global solution and only exploring new states leads to worse overall training results [28]. Especially at the beginning of training, an agent should explore state-action pairs. As the agent has explored a lot it might be good to slowly shift more and more to exploiting the policy. This is exactly what an *ϵ-greedy-policy* does. With a probability of ϵ it chooses a random action out of all possible actions in the given state, and with a probability of $1 - \epsilon$ it chooses the best action greedily as returned by the policy π. Choosing the ϵ defines which way to go between exploration and exploitation, the larger the ϵ the more exploration and vice versa.

4.3 Return

The agent should not rate an action only upon how large the reward of this particular action is but also by how good the possible rewards in succeeding states are. The cumulative sum of all rewards is called the *return* [25] and is defined as

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_{T-1} + R_T$$

where T is the final step. The existence of a final step indicates that there are terminal states from which no further action is possible. We call a trajectory of succeeding states that ends in a terminal state an *episode*. An example of that is a game where the episode ends when the agent either won or lost the game. But sometimes the problem the agent is trained to solve does not have a final state. The return for such a continuous task is defined by

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

with $0 \leq \gamma \leq 1$ [25]. γ is called *discount rate* and determines how future rewards should be weighted. For $\gamma = 0$ only the immediate reward of one action is considered whereas for $\gamma = 1$ all future rewards are considered equally. So the larger γ the more the return represents also rewards further away in the future. Such a discounted return can also be used for an episodic task as can be seen later in this thesis.

4.4 Value-functions

We can now use the return to determine the quality of a state by using a state-value function that is defined as follows.

Definition 6. A *state-value function* is a function $v : \mathcal{S} \to \mathbb{R}$ where $\forall s \in \mathcal{S} : v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$ [25].

Here t is the timestep and $\mathbb{E}_\pi[\cdot]$ describes the expected return when following policy π for the given state s. Similarly to this function determining the expected return of a state we can also define such a function that determines the expected return when taking a specific action a in a given state s and following π afterwards.

Definition 7. An *action-value function* is a function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ where $\forall s \in \mathcal{S}, a \in \mathcal{A} : q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$ [25].

Both of these functions can be transformed to the recursive equations

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big]$$

and

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \sum_{a'} \pi(a',s) q_\pi(s',a')\Big],$$

also called the *Bellman equations* [25] for v_π and q_π. The recursive form shows how the successor states are related to the value of the current state (respectively how the successor state-action pairs are related to the current state-action pair).

A reinforcement learning agent's goal is to find an *optimal policy* [25]. We denote an optimal policy by π_*. When talking about optimal policies we can in the same way also introduce the *optimal value function* [25] and *optimal action-value function* [25] as

$$v_*(s) = \max_\pi v_\pi(s)$$

and

$$q_*(s,a) = \max_\pi q_\pi(s,a).$$

They are defined by simply exploiting the policy which maximizes the return the most. These two functions can be transformed into

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_*(s')\Big]$$

and

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \max_a q_*(s',a')\Big],$$

which are called the *Bellman optimality equations* [25].

4.5 Q-learning with Q-tables

Q-learning is a reinforcement learning algorithm to train an agent in a defined environment. The idea of q-learning is that for each state the agent can determine the quality of every possible action that it can choose.

action / state	A_1	A_2	A_3
S_1	$Q(S_1, A_1)$	$Q(S_1, A_2)$	$Q(S_1, A_3)$
S_2	$Q(S_2, A_1)$	$Q(S_2, A_2)$	$Q(S_2, A_3)$

Figure 4.2: An example for a q-table with two states and three actions

In the tabular version of q-learning, a q-value for each state-action-pair is stored in a table. An example can be seen in Figure 4.2. The q-table can be initialized with random values. In the beginning, these values do not represent the quality of the corresponding actions in any manner. During training, each time the agent chooses an action in a state it receives a reward corresponding to the state-action pair. Now the q-value of

this pair can be updated since the received reward gives information about how good this action is in the given state. The q-values are updated by using

$$Q(S_t, A_t) = (1 - \alpha)\ Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \right) \text{ [25]}. \tag{2}$$

α is called the *learning rate* and describes to what extent the newly calculated value will influence the old q-value. With $\alpha = 1$ it means that the old q-value will be overwritten by a new one such that the old value does not have any influence on the new one. Whereas setting $\alpha = 0$ means that the new observation does not have any influence at all because the q-values always stay the same. Useful values for α lie between these two values. The larger α the more influence new observations have and therefore already learned action values can easily get lost. This is why usually small values are used for α. It leads to a continuous learning curve that is slower, but therefore much more stable. From a q-table a policy can be derived easily by just greedily choosing the action with the largest q-value for each state.
As stated earlier in this section the q-table is initialized with random values. Using a greedy policy that always returns the action with the largest q-value does therefore not lead to a good policy as the largest q-value is just some random value. This is where the exploration-exploitation dilemma (Section 4.2) comes into play. At the beginning of training, exploration should be boosted so that the random values with which the q-table is initialized do not have any influence anymore. While the training is progressing the q-values replicate more and more the true quality of the corresponding state-action pairs. To achieve this the agent uses an ϵ-greedy policy. The key is to start with a large ϵ and decrease it during training from episode to episode. This results in a lot of exploration in the beginning and refining the q-values over time.

4.6 Optimistic Initialization

Instead of using random initialization of the q-table at the beginning, the q-table can be initialized with *optimistic initial values*, which pushes the agent towards more exploration at the beginning of training [25]. Initializing the q-table with optimistic values means to set the hightest possible q-value for every entry in the table. If the hightest value is not known a high value is chosen. High q-values lead to the agent thinking every action is good in the beginning and therefore it is more likely that the agent explores all state-action pairs sooner.

4.7 Multi-agent Q-learning

In *multi-agent reinforcement learning* [21] there is more than one agent interacting with the environment. The agents find themselves in a state and each agent can then choose

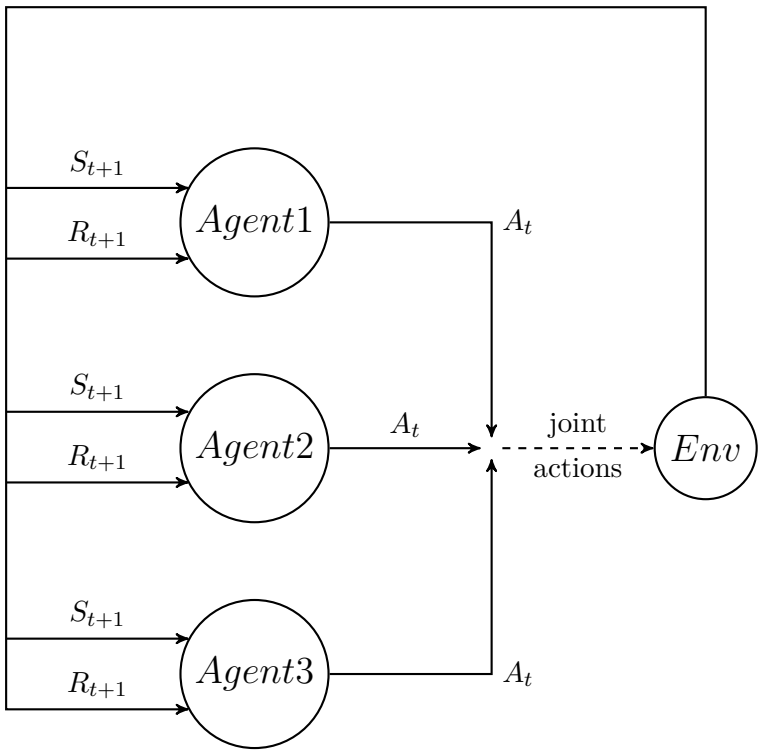


Figure 4.3: Interaction between multiple agents and the environment as in [21]

an action independently of the other agents. The environment then provides one reward to all agents. This reward corresponds to the joint action of all agents' actions together. Additionally, the next state is provided to the agents by the environment. This process can be seen in Figure 4.3 for 3 agents. Since every agent interacts with the environment simultaneously it is considerably harder for each agent to find values that represent each action well. For example, if an agent chooses an action that would have given a high reward in a single-agent setting, it might be possible in the multi-agent setting that another agent simultaneously chooses an action that changes the environment in a way that the first agent's action will be rewarded poorly. In this case, the agent will decrease the q-value of an action that in most cases is a good action to choose.

4.8 Curriculum Learning

Sometimes it can be too hard for the learner to solve a complex problem right away. A huge state space for example can confuse in the beginning because the agent might not be able to recognize good actions. Therefore it might be a good idea to confront the learner with a small subproblem of the actual problem and then gradually increase the difficulty in levels from there. This learning technique is defined in the following.

Definition 8. In *curriculum learning* the goal is to design and choose a full frequence of tasks (i.e. a *curriculum*) $M_1, M_2, \dots M_t$ for an agent to train on, such that learning speed or performance on a target task M_t is improved [20].

A *training level* depicts the training of an agent on one of the tasks in the curriculum. The idea is that the defined tasks for each training level increase in difficulty such that the agent does not have to start to solve a complex task right away. By starting to train the agent to learn a sub-problem of the actual problem the agent can use what it learned in this level to solve the more complex problem in the next training level.
This strategy of defining a curriculum for the learner is derived from how humans and animals learn. Logically it makes sense that the learner will make better progress if the data is not presented in a random fashion but with increasing difficulty. Bengio et al. [4] show this in an example where they train a language model. The model has to learn a vocabulary of defined size. Training inputs are snippets of grammatically correct English sentences of a specific length and the model has to predict the best word that could follow this snippet. The vocabulary is the training distribution, and it is increased from step to step: when training starts the input data will only consist of a vocabulary of size 5,000, but each step it will increase to eventually 20,000. The resulting model is better than a learner that is not trained using a curriculum.
So this strategy applied to a learner can improve the learning process. It can on the one hand lead to faster convergence [14] and on the other hand it can improve the solution quality [4]. The first advantage seems logical: a learner confronted with a very complex problem without having any knowledge about it might need long training to find a solution because there are too many possible ways to try; if the learner has to solve a small subproblem of the initial problem and can use what it learned there to solve problems of increasing size more quickly, it can find a solution more quickly. The idea behind the second advantage is the same: if the agent starts to solve the actual problem immediately without any knowledge it might not find the global solution but only a local one because of the overload of data and possibilities; the global optimum might only be possible to find if the learner figures out how to solve smaller subproblems and how to use this earned knowledge.

5 Applying Q-learning Methods to the Problem

5.1 Construction of the Environment

To construct the environment the number of ports and the number of instructions have to be specified. These define the architecture of the simulated processor. Using this architecture, a port mapping for this processor is generated. So the simulated processor maps instructions to ports according to this port mapping and provides a simulation of the required processing time for given experiments. Furthermore, the environment provides the possibility to create experiments that include the throughput measurements of this processor for the instruction sequences. Lastly, as the environment has to provide a reward for each of the agents' actions, the reward function has to be defined.

5.1.1 Creating the Secret Mapping

The port mapping in the processor that the agent tries to find is called *secret mapping*. As mappings in the three-level-model and therefore the decomposition into μops are not considered in this work, the names of the instructions can be omitted. It suffices to specify the number of instructions and the number of ports. How these instructions are mapped is defined by the corresponding mapping. The secret mapping is created by randomly deciding for each port-instruction pair whether a connection is set or not. We will consider several different constraints when it comes to these connections:

1. no constraints at all,
2. each instruction has at most 75% of the ports as execution option,
3. each instruction has at most 50% of the ports as execution option, and
4. each instruction has at most 25% of the ports as execution option.

Comparing the different outcomes of training for the different constraints helps to analyze if the secret mapping influences the solution quality of the algorithm. An additional constraint that always has to hold for a secret mapping is that each instruction has at least one port as an execution option (as reasoned in Section 2.3). The secret mapping is always denoted with m'.

5.1.2 Creating Experiments

As already explained in Section 2.4, the order of incoming instructions is irrelevant for the experiments. The only adjustable parameter is the experiment length. Given the length of the experiment, random instructions from the instruction set have to be chosen until the experiment has the defined length. While the order does not matter, the frequency of each instruction does: one instruction can be chosen several times. For each experiment, it is simulated how long the processor using the secret mapping needs to execute it. This simulation is done by using the bottleneck simulation algorithm described in Section 2.5. The resulting cycle number is notated next to this experiment so that the agent can provide the reward function the two throughputs as described in the next section.

5.1.3 Defining the Error Function

There are several possibilities when it comes to choosing an error function. In this work we will make use of three different error functions defined by

$$\begin{aligned} R^1(T_1, T_2) &:= -(T_1 - T_2)^2, \\ R^2(T_1, T_2) &:= -|T_1 - T_2|, \\ \text{and } R^3(T_1, T_2) &:= 1 - exp(|T_1 - T_2|). \end{aligned}$$

For each of these error functions, the inputs are 2 throughputs T_1 and T_2 of the same experiment for 2 different mappings. In the following $R \in \{R^1, R^2, R^3\}$ describes any of these error functions, with $R : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}_0^-$. So an error can be negative or 0. An error of 0 means that the throughputs were equal, so 0 means there is no error at all. The greater the difference between the two given throughputs, the worse.

5.2 The Task

Given the set of ports P and the set of instructions I, the agents have to find M such that for all experiments e: $R(t_{m'}, t_m) \leq \delta$ for $m := (\mathsf{I} \dot\cup \mathsf{P}, M)$ with $\delta \in [0, \infty)$. In reinforcement learning often near-optimal policies are found instead of an optimal policy. The optimal case is reached for $\delta = 0$ as this means that the found mapping m processes instructions exactly as fast as the secret mapping m'. Later, it can be seen that this is rarely possible for the agents (especially if the number of ports and the number of instructions is increased). So the task is to find a mapping that fulfills this equation with δ being as small as possible. Even if the found mapping does not reproduce the same throughputs it can still be used for optimization if it resembles the throughputs of the secret mapping as such a small discrepancy is in reality barely noticable.

It should be noted that it is not possible for the agents to check this condition for every possible experiment. Since the length of an experiment does not have an upper bound there are infinitely many possibilities.

5.3 The MDP

Let n be the number of ports and $agent_i$ be the agent corresponding to instruction i. Then the MDP for $agent_i$ is defined by

- the set of states

$$\mathcal{S} := \{P_0, P_0P_1^{a_1}, P_0P_1^{a_1}P_2^{a_2}, \ldots, P_0P_1^{a_1}\ldots P_n^{a_n} | \forall j \in \{1,\ldots,n\} : a_j \in \mathcal{A}\},$$

- the set of actions $\mathcal{A} := \{\text{ON}, \text{OFF}\}$,

- the transition function

$$\mathcal{T}(s_1, a, s_2) := \begin{cases} 1, & \text{if } s_1 = P_0 \wedge s_2 = P_0P_1^a \\ & \text{or } s_1 = P_0P_1^{a_1}\ldots P_m^{a_m} \wedge s_2 = P_0P_1^{a_1}\ldots P_m^{a_m}P_{m+1}^a \\ & \quad \text{with } 1 \leq m < n \text{ and } a_1, \ldots, a_m \in \mathcal{A} \\ 0, & \text{otherwise} \end{cases}, \textit{and}$$

- the reward function

$$\mathcal{R}(s_1, a, s_2) := \begin{cases} 0, & \text{if } s_2 \neq P_0P_1^{a_1}\ldots P_n^{a_n} \\ r(s_2), & \text{if } s_2 = P_0P_1^{a_1}\ldots P_n^{a_n} \end{cases}$$

  with $a_1, \ldots, a_n \in \mathcal{A}$ and $r(s) := R(t_{m'}(e), t_m(e))$ for an experiment e containing instruction i and m being the mapping corresponding to the chosen actions of each agent.

The function r is defined by the error of two throughputs for an experiment. This error can only be calculated if a terminal state is reached as otherwise the mapping m cannot be inferred. After an action for the last port is chosen, the error can be calculated by using function R as defined in Section 5.1.3. Theoretically it is sufficient to have only the information about the set ports for instruction i to calculate the throughput for an experiment that only contains this instruction. Practically the bottleneck simulation algorithm needs the complete mapping, so the final state of each agent. As the agents act independently from each other this is not a problem, it is waited for each agent

to choose actions for all ports to afterwards infer the complete mapping m from each agent's final state. So for each instruction i there is one agent interacting with the environment as defined in the MDP above. Then simulating experiments that contain more than one instruction is also possible.
The error generated for a mapping and an experiment corresponds to all chosen actions for the instructions in the experiment. By using $\mathcal{R}$ as defined above the reward only corresponds to the chosen action for the last port. But the order of these ports is arbitrary, so giving this reward only for the action chosen for the last port does not seem logically. Therefore reward shaping is applied such that each of the chosen actions is rewarded with the error resulting from the experiment. This is done by using the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}' \rangle$ instead, where

$$\mathcal{R}'(s_1, a, s_2) := \mathcal{R}(s_1, a, s_2) + f(s_2)$$

and

$$f_\pi(s_k) := \begin{cases} 0, & \text{if } k = n \\ \sum_{i=k}^{n-1} R(s_i, \pi(s_i), s_{i+1}), & \text{otherwise} \end{cases}$$

with s_{i+1} being the state after choosing action $\pi(s_i)$ in s_i for the current policy π.

So for the action leading to the terminal state, nothing changes, but for preceeding actions the same reward is applied due to function f.
This MDP is defined to solve a combinatorial optimization problem. In reinforcement learning the agent tries to learn which action is best to choose in a given state. As the goal is to find the best mapping, actions are irrelevant after the task is solved. The learner could then also forget the learned q-values as long as the mapping is contained. This means that after training agents in the described environment the resulting agents are not better at inferring other different port mappings. They find a good port mapping, but afterward do not have any advantage in finding another processor's mapping. That is why the MDP is designed in a way that does not seem intuitive at first sight.

5.4 Construction of the Algorithm

The algorithm uses tabular q-learning to train reinforcement learning agents to find the secret mapping of a processor. The agents are trained for a defined number of episodes. In each episode, a mapping is generated and the environment is requested for random experiments. The throughputs of these experiments with the generated mapping and the ones using the secret mapping (noted in each experiment) are used to get a reward from the environment. Using this reward the agent can update the q-table for all chosen actions. This process is visualized in Figure 5.1.

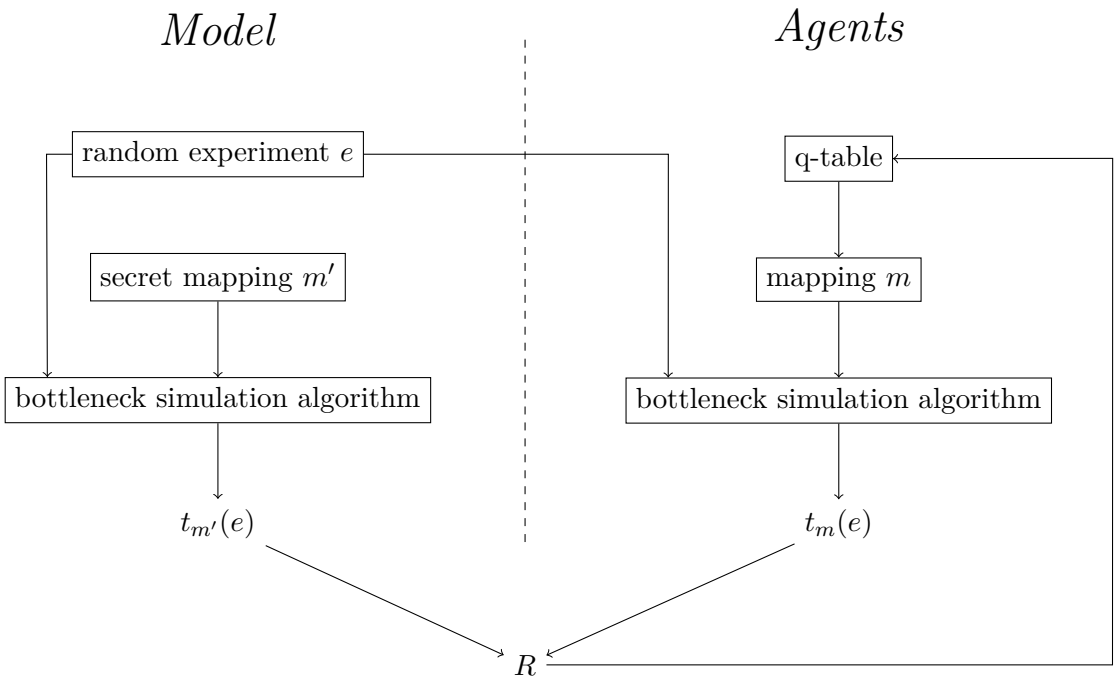


Figure 5.1: Structure of the algorithm

5.4.1 Defining the Q-table

Designing the q-table to find the secret mapping is not straightforward. When constructing the q-table for the state set as defined in Section 5.3 there are $2^{n+1} - 1$ possible states for n ports for each instruction. So the size of the q-table grows exponentially for the number of ports. For example for 20 ports[1] the q-table for one instruction would have size 2,097,151. As there usually are several hundred instructions the computational resource needed would then be even greater.
Therefore a different design is chosen for the q-tables. The goal is to find the mapping, so the interesting states in the MDP are the final states for each agent. These final states depict all permutations that are possible for a port mapping. Choosing only these states as entries in our q-table would also exceed a reasonable limit as the number of states would be half as large[2] as in the previously described approach. The relevant information in each of these final states is, if a certain port connection is set ON or OFF. By extracting only this information without caring about how other ports are set, the size of the q-table can be reduced dramatically. So the states $P_0P_1^{ON}$ and $P_0P_1^{OFF}$ are handled as one state of the q-table, denoted by P_1. In general all states $\ldots P_m^{a_m}$ are handled as state P_m of the q-table independent of which states and actions are chosen before and independent of action a_m. So the states of such a q-table are the different

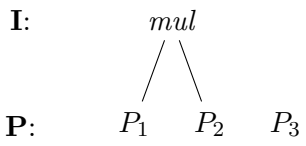
[1] Most of the microarchitectures used nowadays do not have more than 10 ports, but defining an algorithm on this assumption would restrict its usability for the ones that have more.
[2] As there are 2^n final states this would lead to 1,048,576 rows in the q-table

ports, and the actions for each of these states are switching the corresponding port ON or OFF. When using this approach, the q-table has the size *number of ports* $\times$ 2. Such a q-table exists for each instruction because each agent tries to find the mapping for one instruction and therefore needs its own q-table. As the rewards an agent can obtain lie in the range $(-\infty, 0]$, the q-values will also lie in this range (with greater q-values representing better actions). An agent can infer the mapping corresponding to its instruction from the q-values in the table by comparing for each port the two action values for ON and OFF and choosing the larger one.

Example 5.4.1. Let the q-table in Figure 5.2a be the given q-table for instruction *mul*. Then the corresponding mapping inferred from this q-table can be seen in Figure 5.2b. For P_1 and P_2, the q-value for action ON is greater than the one for OFF, so this connection is set. On the other hand for P_3 the q-value for OFF is larger than the corresponding ON value, therefore there is no connection to this port.

	ON	OFF
P_1	-0.5	-0.8
P_2	-0.2	-0.3
P_3	-0.4	-0.1

(a) q-table



(b) port mapping

Figure 5.2: An example q-table and the corresponding mapping

5.4.2 Instructions as Agents

As already described when defining the MDP, for each instruction a seperate agent is defined such that the problem shifts to a multi-agent problem. Each agent is responsible for one instruction, meaning that each agent has a q-table as described in the previous section and tries to find good q-values that represent the connections of the agent's instruction to the ports. For reasons of convenience one can rearrange the values of each agent's q-table and integrate it in one shared q-table. In such a shared q-table the rows are the instructions and the columns for the different ports are in each case divided into one subcolumn for ON and one subcolumn for OFF.

Example 5.4.2. Let an architecture with instructions *add*, *mul*, *store*, and *load*, and 3 ports be given. Figure 5.3a displays the agents' q-tables corresponding to such an architecture. Each agent has its q-table and only cares about one instruction. From all agents' q-tables the full mapping can be inferred as for each instruction the corresponding q-table characterizes which port can be used. The mapping inferred from the 4 q-tables in Figure 5.3a can be seen in Figure 5.3b. Figure 5.3c shows the shared q-table derived from the four agent's q-tables from Figure 5.3a.

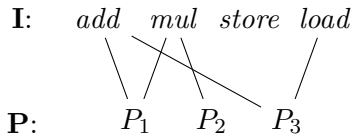
add	ON	OFF
P_1	-0.5	-0.8
P_2	-0.9	-0.3
P_3	-0.4	-0.5

mul	ON	OFF
P_1	-0.5	-0.8
P_2	-0.2	-0.3
P_3	-0.4	-0.1

store	ON	OFF
P_1	-0.9	-0.8
P_2	-0.2	-0.1
P_3	-0.5	-0.1

load	ON	OFF
P_1	-0.5	-0.3
P_2	-0.2	-0.1
P_3	-0.3	-0.5

(a) One q-table for each agent



(b) The corresponding port mapping

	P_1		P_2		P_3	
	ON	OFF	ON	OFF	ON	OFF
add	−0.5	−0.8	−0.9	−0.3	−0.4	−0.5
mul	−0.5	−0.8	−0.2	−0.3	−0.4	−0.1
store	−0.9	−0.8	−0.2	−0.1	−0.5	−0.1
load	−0.5	−0.3	−0.2	−0.1	−0.3	−0.5

(c) One shared q-table for all agents together

Figure 5.3: Example tables and mapping

5.4.3 Updating the Q-table

In the beginning of each episode the mapping m is inferred from the current (shared) q-table. This mapping m is the mapping used in this episode. By inferring a mapping the agents choose actions for all the states: for example if the current q-table would look like the one in Figure 5.3c, $agent_{add}$ would choose action ON in state P_1, OFF in state P_2, and ON in state P_3.
We define exp_{number} as the number of experiments generated and exp_{length} as the maximal length of these experiments. Every experiment is already annotated with the throughput of the processor using the secret mapping m'. For each experiment e the throughput $t_m(e)$ is simulated. Since there are now two throughputs $t_{m'}(e)$ and $t_m(e)$ the reward can be provided by the environment. For each experiment the reward will only be provided to the agents corresponding to the instructions contained in this experiment. If for example the experiment $\{add \to 1\}$ leads to reward -0.5, then only $agent_{add}$ uses this reward to update its q-table, none of the other agents has to update it. That is because the instruction in the experiment only uses the ports corresponding to the actions that this agent chose, so it is completely independent from the other agents' action choices. So for each of this agent's chosen action the corresponding q-value is updated using Equation (2).
This procedure is structured in Algorithm 1. The parameters of this algorithm are the number of ports, the number of instructions, the reward function and the type of initialization of the q-table (random vs. optimistic). These parameters are part of the model and are independent of the algorithm as such. The hyperparameters that adjust the algorithm itself are the number of episodes, the number of experiments used each

Algorithm 1: Multi-agentTraining

Let P be the set of Ports;
Create one q-table for each instruction, initialized randomly/optimistically;
$\epsilon \leftarrow \epsilon_{start}$;
foreach *episode i* **do**
  Infer mapping m from current q-tables ϵ-greedily;
  Generate random experiments E (each containing throughput $t_{m'}$);
  for *experiment e in E* **do**
    Simulate throughput t_m of e;
    $r_e \leftarrow R(t_{m'}, t_m)$;
    foreach *distinct instruction insn in e* **do**
      Let $agent_{insn}$ be the agent corresponding to instruction $insn$;
      for *Port* P_j *in* P **do**
        Let a_j be the chosen action for P_j by $agent_{insn}$;
        $Q_{j+1}^{max} \leftarrow \max\left(Q(P_{j+1}, \text{ON}), Q(P_{j+1}, \text{OFF})\right)$;
        $Q(P_j, a_j) \leftarrow (1-\alpha)\ Q(P_j, On) + \alpha\ \left(r_e + \gamma Q_{j+1}^{max}\right)$;
      end
    end
  end
  if $\epsilon > \epsilon_{end}$ **then**
    $\epsilon \leftarrow \epsilon_{decay} * \epsilon$;
  end
end

episode exp_{number}, the experiment length exp_{length}, the learning rate α, the discount rate γ, ϵ_{start}, ϵ_{decay}, and ϵ_{end}. ϵ_{start} obviously describes the starting value of ϵ. ϵ will be reduced exponentially by multiplying it every episode with ϵ_{decay}. At a certain point, it makes sense to stop this reduction as it can be good to still have some kind of randomness in the action-choosing process. That is what ϵ_{end} specifies: from the episode on when ϵ reaches ϵ_{end} it stays the same till the end of this training level.

5.5 Determining the Mapping Shape in Advance

The *shape* of a mapping describes how many ports are set for each instruction. When using Algorithm 1 to find a port mapping it can happen that the agents find a port mapping which leads to similar throughput times, but does not have the same shape as the secret mapping. Figure 5.4 shows an example: on the left is the secret mapping the agents tried to find, and on the right is the mapping they came up with. The representation of the mappings differs from the one seen earlier in this work. It is the representation used in the implementation, which is also much more convenient for mappings of larger sizes. That is because more ports and instructions lead to more

```
"0": ["2"],
"1": ["1", "2", "3"],
"2": ["0", "1", "3"],
"3": ["0", "1", "3"],
"4": ["0", "2", "3"],
"5": ["0", "1", "2", "3"],
"6": ["3"],
"7": ["2"]
```


(a) secret mapping


```
"0": ["3"],
"1": ["0", "1", "2", "3"],
"2": ["0", "1", "2", "3"],
"3": ["0", "1", "2", "3"],
"4": ["1", "2", "3"],
"5": ["1", "2"],
"6": ["0", "3"],
"7": ["3"]
```


(b) found mapping

Figure 5.4: A secret mapping and the mapping the agents found after training

connections crossing over each other diagonally which can be incomprehensible when looking at. Note that the instructions (and also the ports) are just named after IDs as the name is irrelevant. Port and instruction names start with 0. Each line corresponds to one instruction, namely the instruction at the beginning of the line before the colon. To the right of the instruction, all of its execution options are listed. So in this secret mapping instruction "6" can only be executed on port "3" (whereas for the found mapping the execution options for instruction "6" are ports "0" and "3"). In this example, it is visible that the agents found a mapping that only resembles the shape of the secret mapping, but differs for several instructions. Instructions "1", "2", "3", and "6" have one additional execution option and instruction "5" has 2 instead of 4 ports as an execution option in the found mapping.
To avoid this the shape of the secret mapping can be determined before starting the training. One of the assumptions made for the model we use is that each instruction takes exactly one cycle to be processed by the processor. This property of the model can be used to infer for each instruction how many execution ports exist in the secret mapping. An experiment containing only one instruction can be used as input for the simulated processor. The resulting cycle number reveals how many ports are execution options in the secret mapping. If the resulting number is 1.0, then there is exactly one port as an execution option. If the resulting number is 0.5, then there are two ports as an execution option. So the ports that can be used in the secret mapping for this instruction can be calculated by $\frac{1}{\#cycles}$ with $\#cycles$ meaning the number of cycles the processor using the secret mapping needs to execute this instruction. When doing this for each instruction, the mapping shape can be determined.
With a mapping shape, acting is not straightforward for the agents anymore: if they always choose the actions corresponding to the greater q-values, they might end up setting too many (or not enough) ports to ON. If the mapping shape is known in advance there is no point in inferring a mapping from the q-table that does not match the shape. There are several possibilities to solve this problem. The one that seems to be most intuitive is explained in the following. Given a q-table for an instruction and the number of ports that have to be set, the agent chooses actions of type ON as long as there are the q-values for ON greater than the q-values for the corresponding OFF action. If there are more of these actions available than required, the ones with the greatest q-values are chosen. If on the other hand there are not enough of these actions available,

the agent has to stop to consider only the ON-actions whose q-values are greater than the ones for the OFF-actions. It will then choose the actions which have the greatest q-value for action ON, starting with the greatest and continuing in descending order. In this last choice process the q-values for the corresponding OFF-actions are always greater. The choice process can be seen in Algorithm 2 for one agent.

Algorithm 2: FindActions

```
1  Input: the agent's q-table, the desired number of actions n;
2  Initialize res = {};
3  Let P_ON be the set containing all ports with greater q-values for ON;
4  Let P_OFF be the set containing all ports with greater q-values for OFF;
5  for n steps do
6      if |P_ON| > 0 then
7          p ← arg max_{p∈P_ON} Q(p, ON);
8          add p to res;
9          remove p from P_ON;
10     else
11         p ← arg max_{p∈P_OFF} Q(p, ON);
12         add p to res;
13         remove p from P_OFF;
14     end
15 end
16 return res;
```

Now having the best choices, the random exploration coefficient ϵ has still to be considered. That means that for each action given the result from Algorithm 2, with a probability of ϵ a random value is chosen instead. Such a random value is one of the values that is not in the set containing the best actions so that low q-values get the chance to improve. This process can be seen in Algorithm 3. The result is a port mapping that has the shape as determined in advance by the algorithm. When adding this to Algorithm 1, the mapping shape has to be determined at first before training, and then *inferMappingGivenShape* has to be called in line 5 instead of simply inferring the mapping from the q-tables ϵ-greedily.

5.6 Defining the Training Curriculum

Finding a large port mapping right away can be hard for the agents, so it seems to be a good idea to use curriculum learning and first confront them with smaller problems. Unfortunately, it is not possible to somehow reduce the size of the secret mapping, train the agents to find good q-values for it and then increase the secret mapping's size. That is because the q-values then only fit the mapping of this first training level, they are

Algorithm 3: InferMappingGivenShape

```
Input: the mapping shape;
Let m be a mapping with no connection set;
foreach agent do
    Let n_agent be the desired number of ON-actions as in the given shape;
    P_best = FindActions(Q_agent, n_agent);
    P_rest = P \ P_best;
    for n_agent steps do
        Let num_random be random number between 0 and 1;
        if num_random < ε then
            remove random port from P_rest and set the corresponding connection
              in m;
        else
            remove random port from P_best and set the corresponding connection
              in m;
        end
    end
end
return m;
```

useless for a subsequent training level with increased mapping size (as described in Section 5.3). So another parameter has to be chosen to increase the complexity of each training level when applying curriculum learning to this problem. There are several parameters to tweak in this algorithm with some of them increasing the difficulty of the problem more than others. For example, does the choice of the reward function influence the training process but it is not obvious by looking at different reward functions which makes learning easier for this specific problem. But the maximal length of the experiments that are generated each training episode clearly changes the difficulty for the agents. Each agent is designated for one instruction and tries to find a good port setting for it. By increasing the maximal length of the experiments, the probability that more different instructions are contained in an experiment, increases. That means that a given reward not only corresponds to one agent's chosen actions, but all the agents' chosen actions that correspond to the instructions in the experiment. For example, the throughput of an experiment only containing one *add* instruction will generate a reward corresponding to all the actions $agent_{add}$ chose, so exactly $|\mathsf{P}|$ (for each port either ON or OFF). For each additional distinct instruction in an experiment, the throughput will generate a reward corresponding to $|\mathsf{P}|$ more actions. Therefore it is harder for the agent to find good actions as the provided reward corresponds to more actions when an experiment contains more different instructions.
The generated experiments can vary in length up to the defined maximal length. By increasing the maximal length each level of the curriculum, the probability that experiments with small lengths will occur decreases. If the probability that small experiments are generated is too low the difficulty for the agents could increase too

much to the next training level. Therefore two variations of generating experiments are considered. In the first variation, the different lengths are distributed equally among all generated experiments. In the second variation experiments of smaller lengths are generated more often than larger experiments, so that although experiments get longer from level to level there are still enough small experiments. The first variation is straightforward, so in the following it is explained how the second variation is realized. For the second variation for each experiment a random value is chosen from a list containig each number from 1 up to the defined maximal length. This chosen value defines the length of the experiment. The probability of generating a small experiment is increased by adding more small numbers to this list. For example when sampling a random value from the list $[1, 1, 2]$ the probability that this value is 1 is twice as high as for the value 2. Given the maximal length exp_{length}, the frequency of a given length l in the list is calulated by

$$y(exp_{length}, l) := exp_{length} - (l - 1).$$

This leads to the following sequences (where on the left is the exp_{length} and on the right the list containing each value l exactly $y(exp_{length}, l)$ times):

$$\begin{aligned}
&1 \rightarrow [1] \\
&2 \rightarrow [1, 1, 2] \\
&3 \rightarrow [1, 1, 1, 2, 2, 3] \\
&4 \rightarrow [1, 1, 1, 1, 2, 2, 2, 3, 3, 4] \\
&5 \rightarrow [1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 5] \\
&\vdots \qquad\qquad\qquad\qquad \vdots
\end{aligned}$$

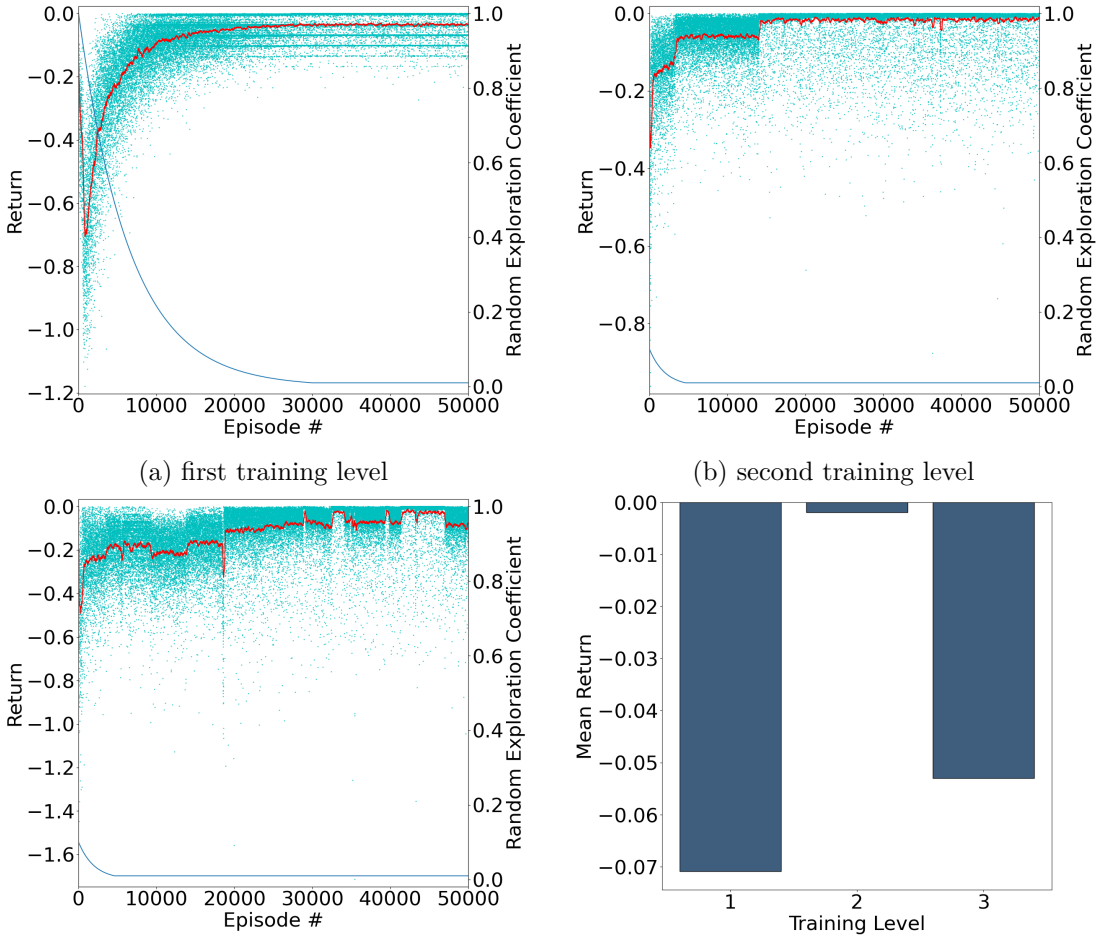
So the probability that small values are chosen does not decrease too much when increasing the maximal length. This second variation is referred to as an unequal experiment distribution whereas the first one is an equal experiment distribution. Experiment distribution is in the following referred to as exp_{distr}.
The key of curriculum learning consists of the fact that the learner can use the insights it learned in the last level to solve the given problem in the next level. That means ϵ_{start} should not be set too high because with a high ϵ_{start} the learner chooses random actions all over again and might change q-values of actions that already converged to a good value. In such a case the results of the previous training levels can get lost. Therefore it is a good idea to set ϵ_{start} to a low value, but not too low so that there is still a small amount of exploration in the beginning.

6 Improving and Analyzing the Algorithms

In this chapter the algorithms presented in the last chapter are analyzed. They have a lot of hyperparameters that can be adjusted. Different values for these are evaluated and compared to each other. After the hyperparameter search it is analyzed how much the secret mapping influences the solution quality of the algorithm. Algorithm 1 is called the base algorithm in the following. The second algorithm is Algorithm 1 inferring the mapping shape in advance, so extended by Algorithm 2 in line 5. This algorithm is called shape-acting algorithm. Before comparing these two algorithms the base algorithm is analyzed in the succeeding section.

6.1 Increasing Solution Stability

To analyze how the algorithm performs in general, some initial values are chosen for the different (hyper)parameters. The number of ports is set to 4 and the number of instructions to 8. These are low values but it is sufficient to depict the procedure of one curriculum and the issues that can occur and how they can be improved. ϵ is set to 1.0, ϵ_{end} is set to 0.01, and ϵ_{decay} is chosen in such a way that ϵ reaches ϵ_{end} after 60% of the episodes of the first training level. The number of episodes is set to 50,000 for every level. This leads to an ϵ_{decay} of ~ 0.99985. R^1 is chosen as the reward function. The q-table is initialized randomly. α is set to 0.002 and γ is set to 0.99. Thirty experiments are generated for each training episode, with a maximal length of 2 in the first, 5 in the second, and 8 in the third level, so all in all 3 levels of training. After each level, an evaluation run with 400 episodes is executed. During each episode of such an evaluation run, random experiments with a maximal length of 3 are generated. These experiments are evaluated with the mapping inferred by the current q-table and are then compared to the actual throughput. For this evaluation the same reward function is used as for the training. So the resulting plot shows how good the current q-values are. Since there are 30 random experiments in each of the 400 episodes (altogether 1200 experiments), it is very unlikely that the agents find a mapping that is only equal for these specific experiments and different for other ones. That is the reason why the training is stopped if an evaluation run's results are only rewards with the value 0.
Figure 6.1 shows 4 plots: 3 training plots of the 3 levels and 1 bar plot showing the mean return for the evaluation run after the corresponding levels. The red line in a plot displays the return. The left y-axis depicts the return scale, the right one the ϵ scale. The return for a single episode is the average of all returns of the generated experiments of this episode. It is visualized by a small dot in the plot in cyan. In the bar plot a bar depicts the mean of the returns of one evaluation run. As there are 3 training levels there are 3 bars. Since the return scale is negative smaller bars stand for better returns.



(a) first training level

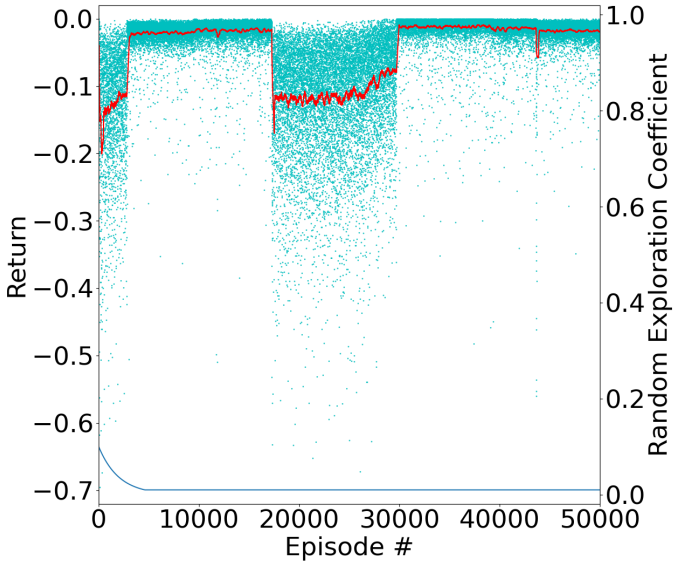
(b) second training level

(c) third training level

(d) means of the evaluation runs

Figure 6.1: Plots of a training run with 3 training levels

In the first training level, ϵ starts larger than in the remaining two. As described in Section 5.6 the initial value for ϵ_{start} should not be too large for other training levels than the first one. In this case, ϵ_{start} is set to 0.1 with ϵ_{decay} being 0.9995. The first training plot shows that the agents get better at finding a good setting as the reward curve is rising towards 0. In the second level of training one can see that the agents got even better than in the first one. This can also be seen in the corresponding bar in the bar plot as the mean of the returns of the found mapping is close to 0. In the third level of training however the agents got worse again as the third bar in the bar plot is showing. The mapping the agents found and the secret mapping is shown in Figure 5.4. It can be seen that the agents managed to find a mapping that kind of resembles the shape of the secret mapping but is different for several instructions.
Unfortunately it does not happen all the time that the agents make steady progress throughout all training steps. What sometimes happens during training is that in the first level the agents learn very solidly, but in one of the succeeding levels, the average reward jumps abruptly to much higher or lower values. This phenomenon can be seen



# ports :	4
# instructions :	8
ϵ_{decay} :	~ 0.9995
ϵ_{end} :	0.01
reward function :	R^1
α :	0.002
γ :	0.99
optimistic init. :	$False$
exp_{number} :	30
exp_{length} :	$2, 5, 8$
mapping constraints :	1.0
training levels :	3
exp_{distr} :	$equal$

Figure 6.2: Training plot of the third training level showing this phenomenon

in in the third training plot in Figure 6.1 or more clearly in Figure 6.2. It visualizes the challenge the agents have to solve. A small change to some instructions' settings can have a big influence on the outcome. If such a change only leads to worse results the q-values are adjusted correspondingly. In such a case the agents learn that the chosen actions are bad. Such a collapse of the reward does not happen by choosing actions that are immediately punished. This happens because choosing different ports for an instruction can lead to better throughput times (and therefore higher reward) for certain experiments, while for other experiments containing different instruction combinations the choice leads to a worse throughput. If an agent already has the correct setting for the corresponding instruction the choices can still be punished with bad rewards due to bad settings of other instructions. That is the reason why it is particularly hard for each agent to find a good setting. They have to come up with a solution that also harmonizes well with the setting of the other agents.
As can be seen in Figure 6.1 just before the end of the last training level the reward curve drops down a little bit. This means that the final q-table does not represent the best mapping the agents found. To make sure that this does not ruin the end result when happening towards the end of training the best mapping during each training step is saved. For that the quality of the current mapping has to be determined regularly. The quality is determined using the *Mean Absolute Percentage Error* (MAPE). Every 100 episodes, 200 random experiments with a maximal length of 10 are generated. Using the current mapping of this episode the MAPE of this mapping is calculated. In this way it can always be checked if the current mapping is better than the best mapping saved to that point by comparing this freshly calculated MAPE to the saved MAPE of this best mapping. The experiment number and experiment length to determine the MAPE are different from the ones used for the evaluation runs after each training

level. For an evaluation run 30 experiments with a maximal length of 3 are generated 400 times. This is because an evaluation run serves as a visualization of how good the current q-table performs, so it is not really important if it is not perfectly accurate. On the other hand when the MAPE is calculated during the training to save the best mapping it should be as accurate as possible because it determines the result of the algorithm. But this calculation also suffers from a trade-off. Ideally the MAPE would be calculated every episode of training and with many experiments. Unfortunately this is not feasible, which is the reason why it is only calculated every 100 episodes for 200 experiments of length 10.

6.2 Mapping Shape in Advance

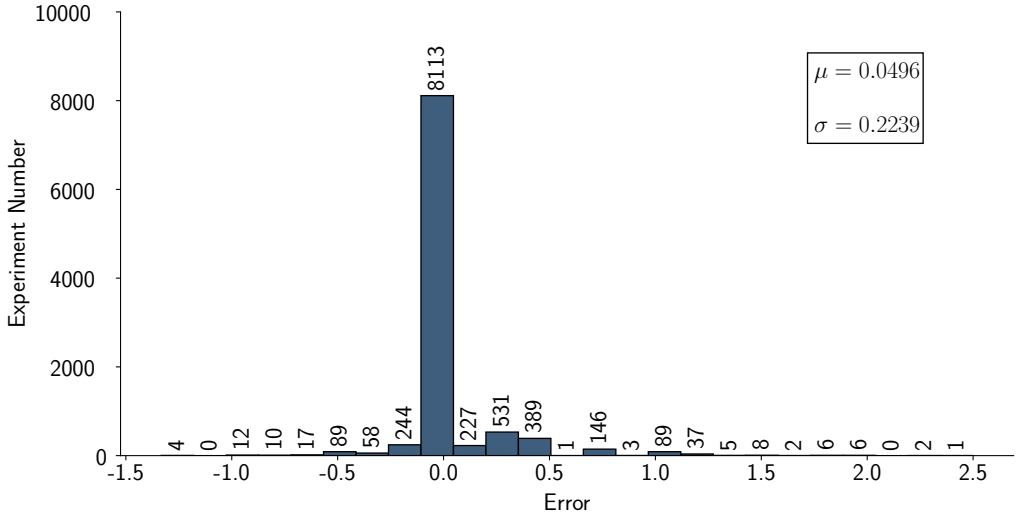
In this section the impact of determining the shape of the secret mapping prior to training is analyzed. So as already stated there are two versions being compared, the base algorithm and the shape-acting algorithm. For the training runs some values have to be chosen for the (hyper)parameters. These can be seen in Figure 6.3. Increasing the number of ports slows down the algorithm significantly as simulating an experiment takes longer. When increasing the total number of instructions it is not as much slower as when increasing the number of ports, but it still slows down the overall running time (this is due to the bottleneck simulation algorithm). For this reason only 4 ports and 40 instructions are set for the parameter search as otherwise the required time would be out of the scope of this thesis. In the evaluation chapter it is analyzed how well the algorithm with the found parameters performs for a setting with larger port and instruction numbers.

# ports :	4	**mapping shape** :	_____
# instructions :	40	optimistic init. :	$False$
ϵ_{decay} :	~ 0.9998	exp_{number} :	30
ϵ_{end} :	0.01	exp_{length} :	$2, 5, 8$
reward function :	R^1	mapping constraints :	1
α :	0.002	training levels :	3
γ :	0.99	exp_{distr} :	$equal$

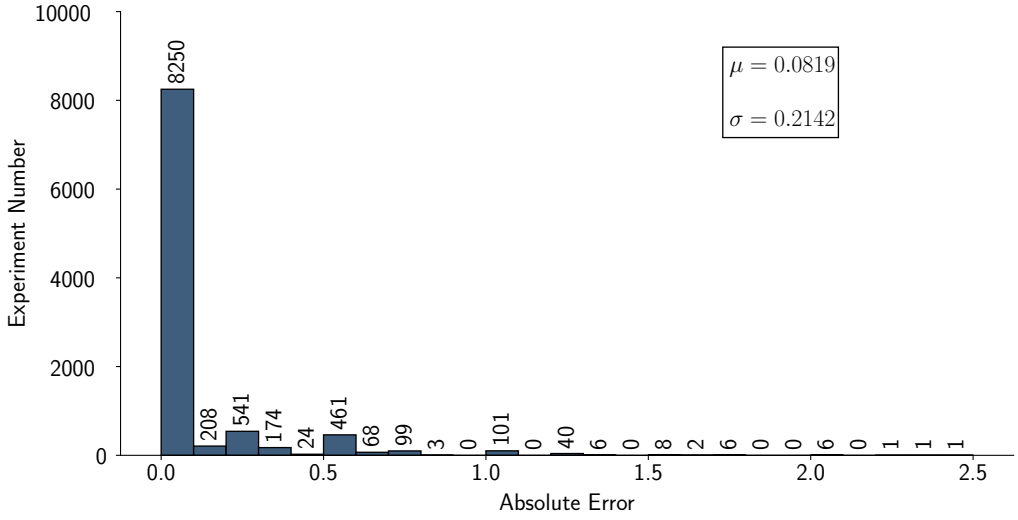
Figure 6.3: Parameter setting with mapping shape being the unfixed parameter

For each of the two versions the algorithm is run 65 times. Each run uses a seperate random seed so that the secret mapping is different in every run. Each of the resulting mappings is used to evaluate 10,000 random experiments. By evaluating these experiments the MAPE of the different throughputs can be calculated, resulting in 65 MAPEs

for each version of the algorithm. These MAPEs can then be compared to see which of the algorithms is better. The number 65 is arbitrary. Due to reasons of limited time and computational resource it is not possible to do more than 65 runs per setting. In Figure 6.4 two different histograms can be seen visualizing these errors. The first histogram depicts for different error ranges how many of the 10,000 experiments fall into the corresponding range. Negative errors mean that the processor using the found mapping took longer for the experiment than the one using the secret mapping, and for positive errors it is the other way around. In the second histogram the absolute error is depicted in the same way. As can be seen in the histograms 10,000 experiments are sufficient to see how well the mapping performs.



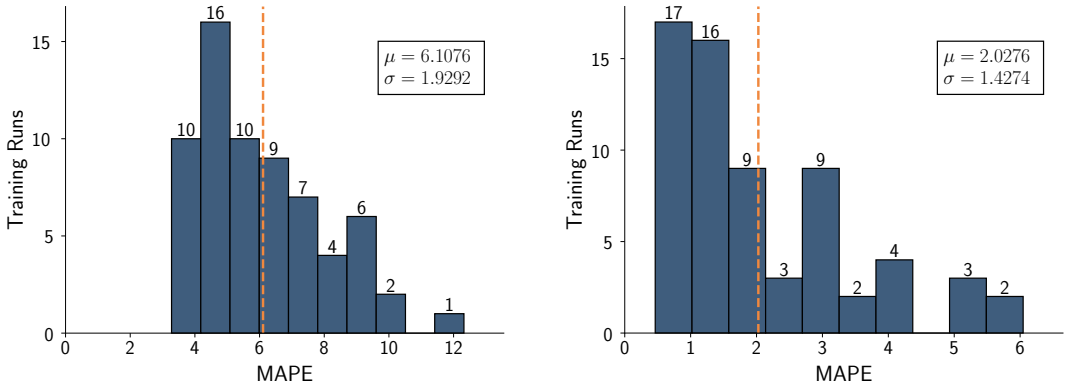
(a) histogram depicting the difference between the throughputs



(b) histogram depicting the absolute difference between the throughputs

Figure 6.4: Histograms visualizing the performance of the found mapping

The distribution of the MAPEs of the evaluation of the 65 runs for both versions can also be visualized in a histogram. Each resulting mapping of these 106 runs is evaluated with 10,000 random experiments as previously explained, and thus there are 65 MAPEs for each version. The two histograms showing the distribution of 65 MAPEs for each version can be seen in Figure 6.5. The vertical orange line depicts the mean. The one on the left corresponds to the base algorithm and the one on the right to the algorithm extended by inferring the mapping shape in advance and acting correspondingly. The difference between the two distributions is clearly visible. The algorithm finds better solutions when inferring the mapping shape in advance as the mean of the MAPEs with determining the mapping shape is 2.0, whereas without it it is 6.1. The standard deviation is also much smaller for the version that determines the mapping shape in advance. So the algorithm performs better if the mapping shape is known to the agents before training.

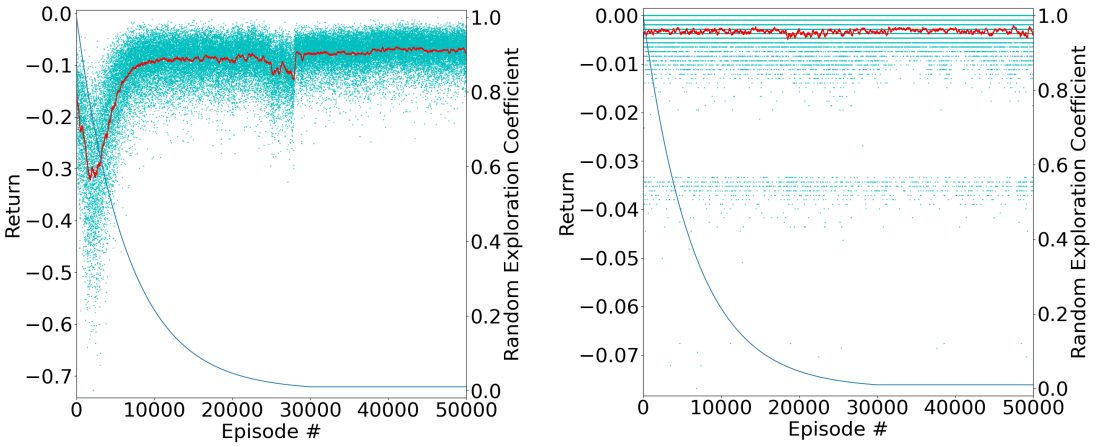


(a) base algorithm

(b) shape-acting algorithm

Figure 6.5: Histograms showing the MAPEs' distribution for both versions

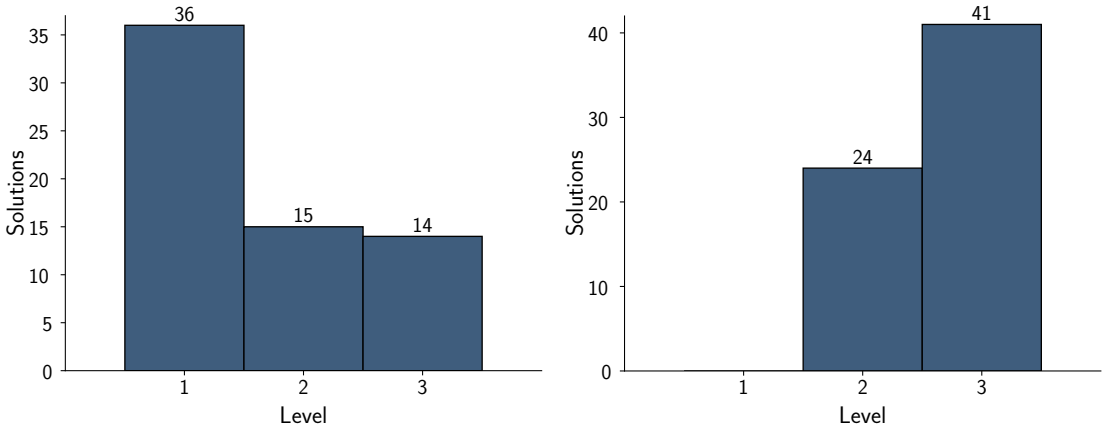
Figure 6.6 shows two training plots: on the left, the graph of the first training level without using the mapping shape (base algorithm) and on the right, the graph of the first training level with using the mapping shape. Each of these is from one of the corresponding 65 training runs. Showing the plots of these two runs particularly is arbitrary, but the plots of the first training level for the other runs look similar to these, so these two serve as a representation. It can be seen that while the base algorithm can improve its q-values during training, the reward curve for the algorithm that infers the mapping shape in advance looks quite disappointing. Nevertheless when looking at the return scales it can be noticed that the base algorithm never reaches as good rewards as the shape-acting algorithm. So when the mapping shape is inferred in the beginning the algorithm immediately infers better mappings. Because this return is already very good it has then a hard time to improve the q-values, especially in this training level with the same experiment size.
It is also interesting to analyze in which training level the agents found the best mapping. Figure 6.7 depicts for each training level how often the best mapping was found in this level. The left histogram is for the version without determining the mapping shape in



(a) base algorithm (b) shape-acting algorithm

Figure 6.6: Two training plots of the first training level

advance and on the right the one that determines it before training. It can be seen that without the mapping shape for most of the runs the best mapping is found within the first training level. So the agents do not always get better from level to level. When the mapping shape is determined it is the other way around as for most of the trainings the best mapping is found in the last training level. What is noticable here is that out of all of the 65 trainings not a single time the best mapping is found in the first level. This suggests that the agents usually make progress from level to level when the mapping shape is known. Further it encourages to check if the agents learn better when there are more levels.



(a) base algorithm (b) shape-acting algorithm

Figure 6.7: Histograms showing in which training level the best mapping is found

6.3 Hyperparameter Search

To find good values for the hyperparameters a partial grid search is performed. A normal grid search consists of searching the best values for all hyperparameters independently of the dependencies between them. Doing this is out of the scope of this thesis. In the following grid search, all hyperparameters except for one are set to some fixed value while different values for the remaining (unfixed) hyperparameter are used to evaluate the different solutions. Though after fixing the found value it will not be cross-validated again later. Doing this partial grid search for every single hyperparameter would also be out of the scope of this thesis. The hyperparameters being analyzed are the number of training levels and the corresponding experiment length exp_{length}, the number of experiments exp_{number} that are generated each episode, the discount factor γ, and the experiment distribution exp_{distr}. So the ones not being analyzed are ϵ, α and the number of episodes. While hyperparameters like exp_{length} are very specific and only exist in this algorithm, ϵ, α, γ and the number of episodes are hyperparameters that are a classical component of a q-learning algorithm. That is the reason why there exists work that specializes on finding good values for them. For example Franklin Fernandez and Wouter Caarls propose a method that uses evolutionary computing techniques to find good values for ϵ, α, and γ for q-learning [9]. Although γ is also analyzed there, it is still evaluated in the following grid search. That is because the theory of the MDP changes when setting γ to 0.0. By setting it to 0.0, the next port is not influencing the new q-value when it is updated because the second part of the last summand in equation (2) is 0. So then the actions the agents choose are independent from each other. Therefore it is interesting to see how the solution quality of the algorithm changes when changing the value of γ.
The evaluation concept applied in the last section to compare the different versions of the algorithm is also used in this partial hyperparameter search. So for each parameter under test the algorithm is run 65 times and each of these runs is evaluated using the error between the throughput of the resulting mapping and the one of the secret mapping for 10,000 experiments. When analyzing the next hyperparameter, the 65 runs of the chosen setting can be reused as one of the settings for the next hyperparameter under test.

6.3.1 Number of Training Levels & Experiment Length

In this section it is analyzed how the algorithm performs when it is run with a different number of training levels. Changing the number of training levels comes along with defining the experiment length for each of these levels. That is because for each additional level of training the experiment length has to be specified. Therefore both parameters are changed at once in this step of the grid search. Figure 6.8 shows the fixed parameters.

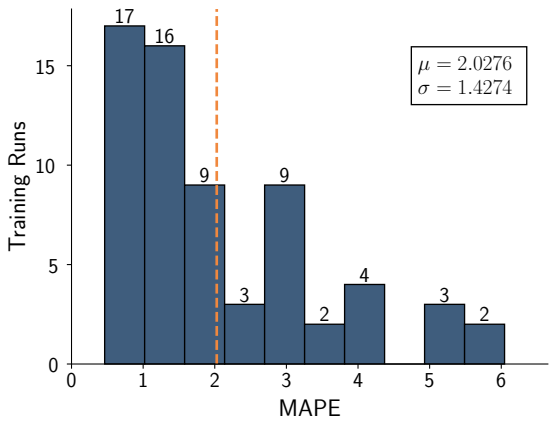
# ports :	4	mapping shape :	True
# instructions :	40	optimistic init. :	$False$
ϵ_{decay} :	~ 0.9998	exp_{number} :	30
ϵ_{end} :	0.01	exp_{length} :	_____
reward function :	R^1	mapping constraints :	1
α :	0.002	**training levels** :	_____
γ :	0.99	exp_{distr} :	*equal*

Figure 6.8: Parameter setting with exp_{length} and the number of training levels being the unfixed parameters

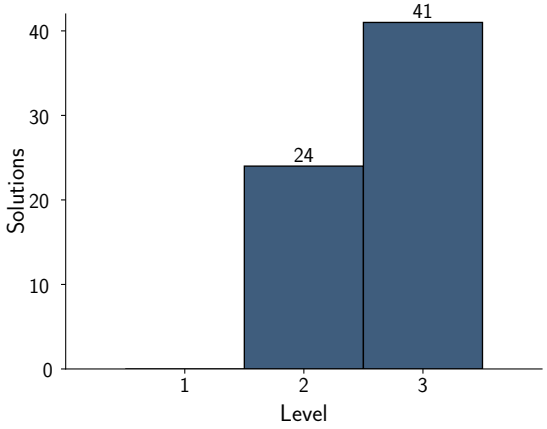
The algorithm is analyzed for three different settings:

- 3 training levels with experiment lengths 2, 5, and 8
- 4 training levels with experiment lengths 2, 4, 6, and 8
- 7 training levels with experiments lengths 2, 3, 4, 5, 6, 7, 8

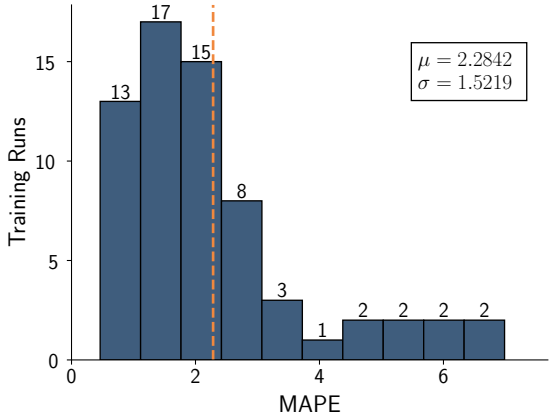
The first of the three settings corresponds to one of the settings from the last section, so these 65 runs are used again for this comparison. The three histograms showing the distribution of the the MAPEs of the 65 runs for each parameter variation can be seen in Figure 6.9. All three of them are quite similar: the means of these MAPEs are 2.03, 2.28 and 2.05, so fairly close together.
For these three settings it might be particularly interesting to see in which level the best mapping is found due to the different level settings. Figure 6.9 depicts the corresponding distributions. The histograms show that by using more levels with experiment lengths in between the 3-level version the best mapping is often found earlier. But as seen in Figure 6.9 these mappings are not better than the ones found late in training runs with fewer levels. The spots when they are found are just stretched among the additional levels, with the number of found mappings rising from level to level.
Since the results of the 4-level variant is worse than the other two choosing this setting would not make sense. The results of the 3-level variant and the 7-level variant are almost equally good, therefore the 3-level variant is chosen from now on. Using the other variant would mean to sacrifice computational resource as it takes more time to train the agents 4 more levels.



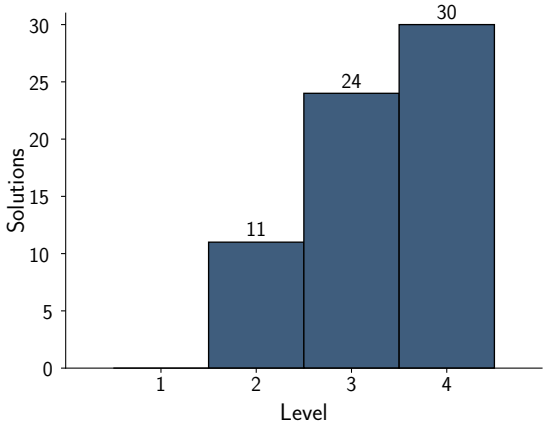
(a) 3 training levels



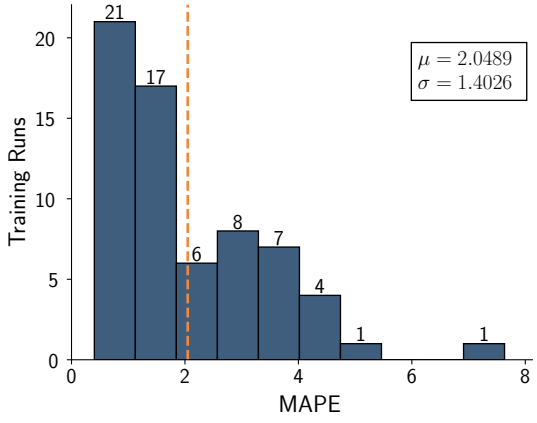
(b) 3 training levels



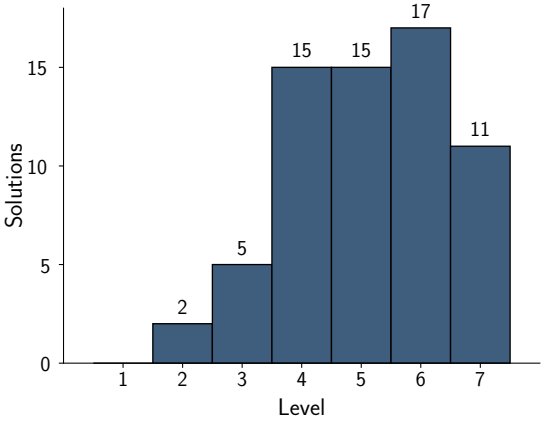
(c) 4 training levels



(d) 4 training levels



(e) 7 training levels



(f) 7 training levels

Figure 6.9: On the left the distribution of the MAPEs of 65 runs for different training levels, on the right in which training level the best mapping is found

6.3.2 Number of Experiments per Episode

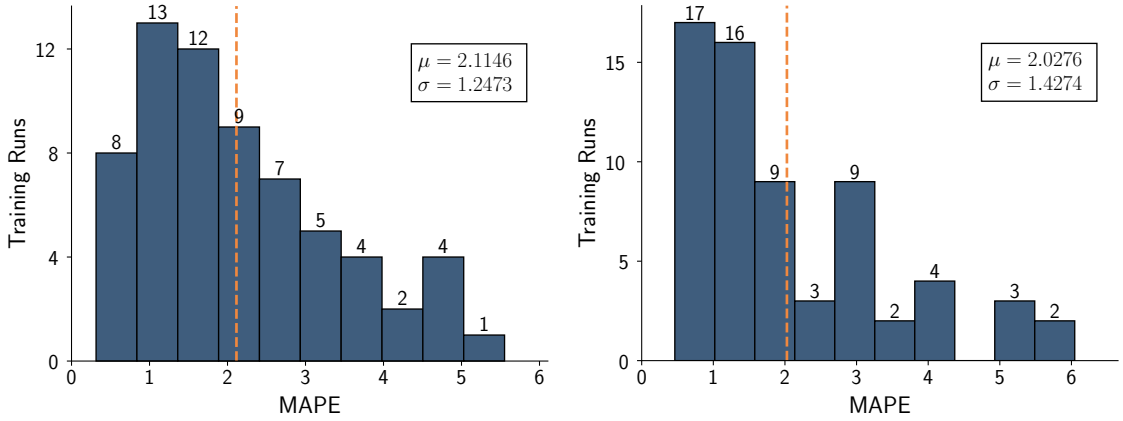
The next parameter to be analyzed is exp_{number}. It defines how many experiments are generated each training episode to generate rewards. As concluded in the last section three levels of training are used with the experiment lengths 2, 5, and 8. A full list of the parameter setting is depicted in Figure 6.10. For exp_{number} three different values are used to analyze how it influences the performance of the algorithm. The used values are 20, 30, and 40. The algorithm is run 65 times for each of these settings. For the setting with $exp_{number} = 30$ the 65 runs from the previous section are used again. Again, each of these 195 runs is evaluated by 10,000 random experiments, generating an MAPE.

# ports :	4	mapping shape :	True
# instructions :	40	optimistic init. :	$False$
ϵ_{decay} :	~ 0.9998	exp_{number} :	———
ϵ_{end} :	0.01	exp_{length} :	$2, 5, 8$
reward function :	R^1	mapping constraints :	1
α :	0.002	training levels :	3
γ :	0.99	exp_{distr} :	$equal$

Figure 6.10: Parameter setting with exp_{number} being the unfixed parameter

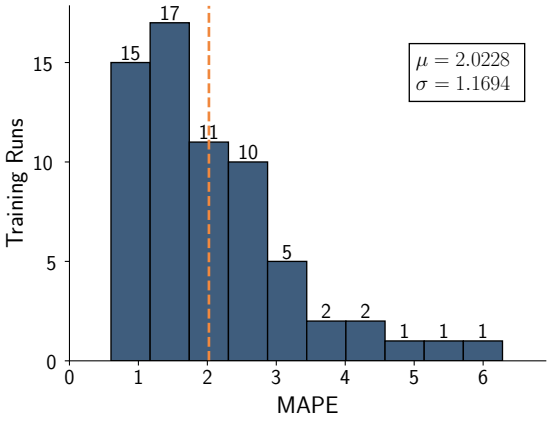
The distribution of the resulting MAPEs is depicted in the histograms in Figure 6.11. It can be seen that changing the number of experiments that are generated each episode does not have much influence on the solution quality, the mean MAPE is about the same for all three versions. This is very interesting as for the setting with $exp_{number} = 40$ twice as many experiments are evaluated each episode than when setting exp_{number} to 20. That means twice as many experiments are evaluated in total over the coarse of 50,000 episodes, for each training level. Despite these additional rewards the agents are not significantly better at finding the mapping. This could be out of the same reason why more episodes could also not be an improvement: at a certain point the agents cannot learn more than they already have in this particular level, no matter how many episodes are added. Increasing the experiment number is not the same as increasing the total number of episodes as in the former the random exploration coefficient is higher for a part of the added training time. Nevertheless it is similar. Another reason that the agents do not improve despite this additional training time could be that the problem is too complex to solve it with tabular q-learning. It could be that the agents are simply not able to find a better mapping than one that still has a small error.
The mean of the setting with $exp_{number} = 20$ is worse than the means of the other two settings. Therefore this setting is not chosen. The standard deviation of the setting with $exp_{number} = 40$ is better than the one for the setting with $exp_{number} = 30$, however the means are about equal. Because the means are about equal and simulating more

experiments each episode takes more time, the 30-experiment variant is chosen as the resulting best setting. So for the next runs, exp_{number} is set to 30.



(a) $exp_{number} = 20$

(b) $exp_{number} = 30$



(c) $exp_{number} = 40$

Figure 6.11: The distribution of the MAPEs of 65 runs for the 3 parameter settings

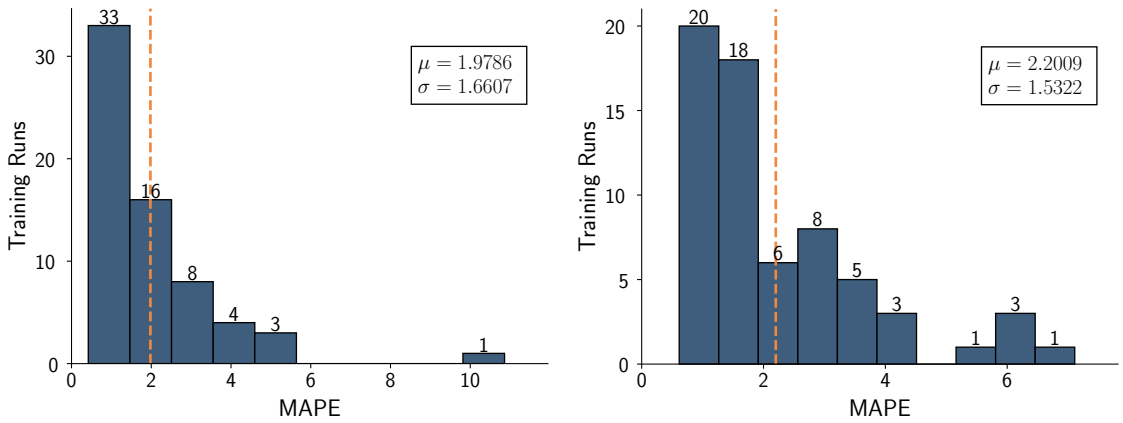
6.3.3 Discount Factor

In the beginning of this chapter the motivation for checking the algorithm's performance with γ set to 0.0 is explained. Additionally to that a third version is run, where γ is set to 0.8, to see if something in between could be better than one of the two edge cases. Again, 65 runs for each of the 3 versions are compared by evaluating them with 10,000 experiments and the corresponding MAPEs. The parameter setting can be seen in Figure 6.12.
Figure 6.13 shows the histograms of the distribution of the MAPEs for the different settings. It can be seen that for $\gamma = 0.0$ and $\gamma = 0.99$ the means are almost equal, for $\gamma = 0.8$ it is a bit worse. This suggests that the algorithm is stable with regards to γ. That is probably because the trajectories are not very long, especially with this setting

# ports :	4	mapping shape :	True
# instructions :	40	optimistic init. :	$False$
ϵ_{decay} :	~ 0.9998	exp_{number} :	30
ϵ_{end} :	0.01	exp_{length} :	$2, 5, 8$
reward function :	R^1	mapping constraints :	1
α :	0.002	training levels :	3
γ :	_____	exp_{distr} :	$equal$

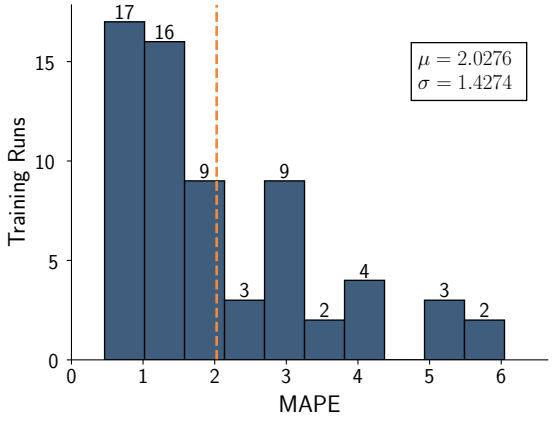
Figure 6.12: Parameter setting with γ being the unfixed parameter

with only 4 ports. But even when using 10 ports the trajectories are still short, that could be the reason why γ does not have too much influence on the result. As the mean of the setting with $\gamma = 0.0$ is only by 0.05 smaller but the standard deviation of the setting with $\gamma = 0.99$ is by 0.23 smaller the chosen value for γ is 0.99.



(a) $\gamma = 0.0$

(b) $\gamma = 0.8$



(c) $\gamma = 0.99$

Figure 6.13: The distribution of the MAPEs of 65 runs for the 3 parameter settings

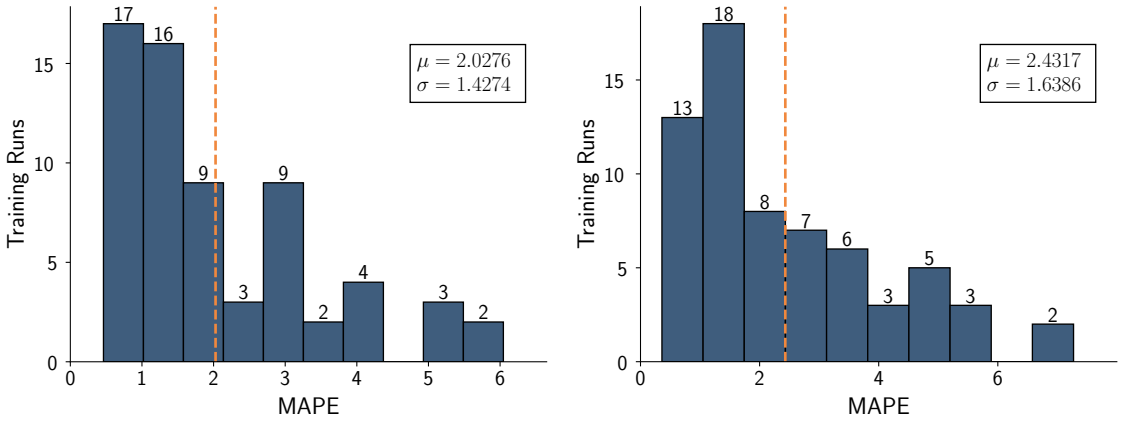
6.3.4 Experiment Distribution

The last hyperparameter that is analyzed in this grid search is exp_{distr}, i.e. how the length of the chosen experiments is distributed. The two settings for this hyperparameter are either an equal experiment distribution or an experiment distribution as described in Section 5.6, denoted as unequal. The parameter setting is shown in Figure 6.14.

# ports :	4	mapping shape :	True
# instructions :	40	optimistic init. :	$False$
ϵ_{decay} :	~ 0.9998	exp_{number} :	30
ϵ_{end} :	0.01	exp_{length} :	$2, 5, 8$
reward function :	R^1	mapping constraints :	1
α :	0.002	training levels :	3
γ :	0.99	exp_{distr} :	_____

Figure 6.14: Parameter setting with exp_{distr} being the unfixed parameter

Figure 6.15 shows the distribution of the resulting MAPEs. It can be seen that with unequal experiment distribution the agents do not learn as well as when the length of the experiments are distributed equally. The intention of the distribution as defined in Section 5.6 is to avoid that the difficulty when transitioning to the next level is increased too quickly. The resulting histograms show that the opposite is the case: too many small experiments seem to slow down the learning progress such that the agents eventually come up with worse results. Therefore the equal distribution is the better choice for this hyperparameter.



(a) $exp_{distr} = equal$ (b) $exp_{distr} = unequal$

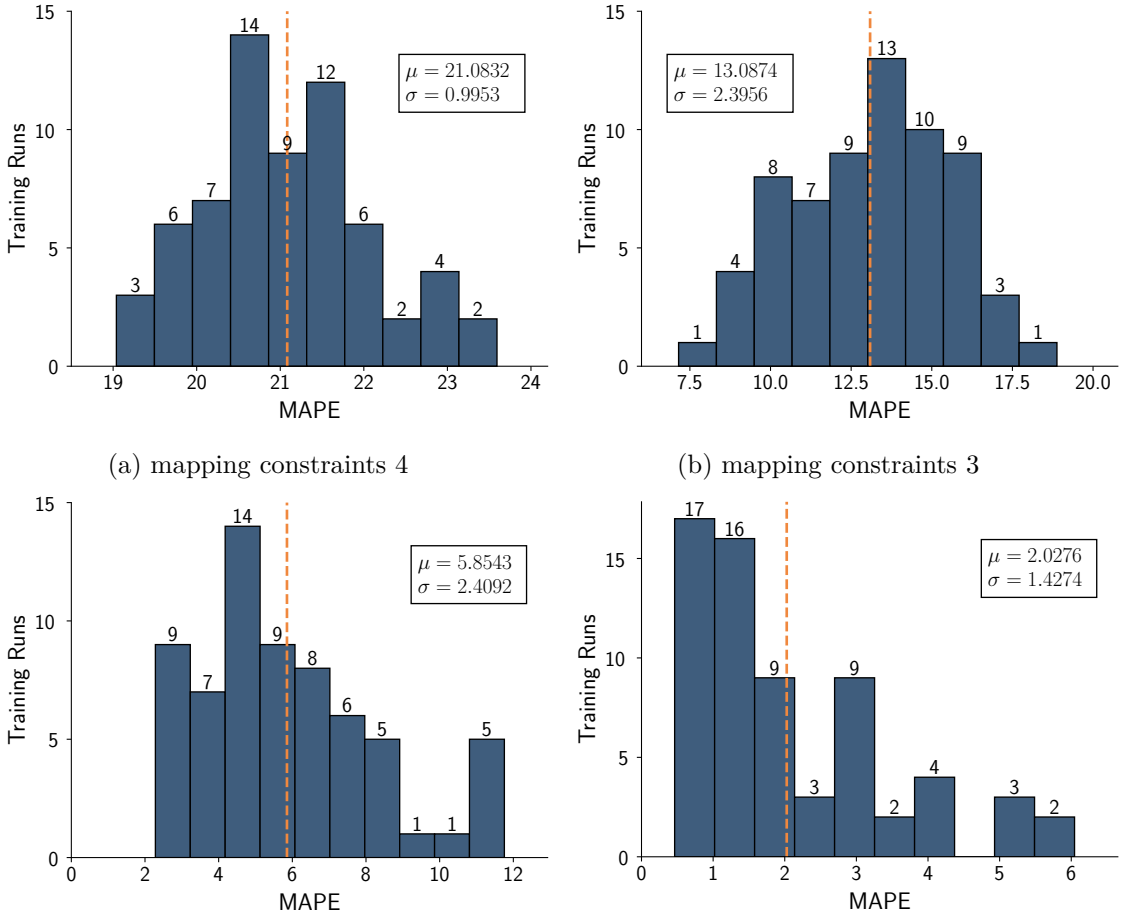
Figure 6.15: The distribution of the MAPEs of 65 runs for the two parameter settings

6.4 Influence of the Mapping

In this section the influence of the mapping on the outcome of the algorithm is analyzed. As described in Section 5.1.1. 4 different mapping constraints are studied:

1. no constraints at all,
2. each instruction has at most 75% of the ports as execution option,
3. each instruction has at most 50% of the ports as execution option, and
4. each instruction has at most 25% of the ports as execution option.

For each of these constraints 65 random mappings are generated satisfying the corresponding constraint. Then the algorithm is run to train the agents to find these mappings. The found mappings are then used to evaluate 10,000 experiments, leading to 65 MAPEs per constraint on the secret mapping. Figure 6.16 shows the distribution of these MAPEs for the 4 constraints.
It can be seen that these constraints have a big impact on the solution quality of the algorithm. The solution quality improves if the constraints are reduced. For a setting with 4 ports constraint 4 leads to setting exactly one port per instruction to ON (0 ports are not allowed as there has to be an execution option for each instruction). For this kind of setting it seems very unintuitive that the algorithm does not find a good solution. Nevertheless the reason is quite straightforward. Since the mapping shape of the secret mapping is inferred before the training experiments of length 1 result in a reward of 0.0 for any random mapping the agents come up with. In the first level only experiments of length 1 and 2 are generated. As there is only one port set for a setting with 4 ports and constraint 4 the rewards can only be -1.0 or 0.0 in this training level. The fewer constraints are set the more rewards are possible in between these two numbers, leading to better results. Because R^1 is used rewards of -1.0 are considerably worse than rewards between 0.0 and -1.0, therefore the reward gets better for fewer constraints. So as more execution options are possible for each instruction the possibility that the throughputs are closer although not the correct mapping is found increases. The more constraints for the secret mapping the worse the performance of the algorithm.



(a) mapping constraints 4

(b) mapping constraints 3

(c) mapping constraints 2

(d) mapping constraints 1

Figure 6.16: The distribution of the MAPEs of 65 runs for the 4 different mapping constraints

7 Evaluation

Lastly, the algorithm has to be evaluated to see how well it can perform for mappings of a larger size, which is the focus of this chapter. As good values for the hyperparameters are found the remaining parameters have to be evaluated. It is particularly interesting to see how the algorithm performs for larger mappings as then the complexity of the task increases. The used setting for this evaluation is 8 ports and 310 instructions. These numbers are chosen because Ritter and Hack [24] evaluated their algorithm using this setting. It has to be said that their algorithm is not comparable to the one of this thesis as they infer 3-level-mappings and thus it is important which instructions they choose. In our approach instruction types are irrelevant. It is interesting to see how the algorithm performs for considerably larger mappings and therefore these numbers (310 instructions, 8 ports) are taken over.
The reward functions being analyzed are the three functions defined in Section 5.1.3. Additionally, it is analyzed how using another initialization method changes the solution quality of the algorithm. The two initialization methods for the q-table that are evaluated are random initialization and optimistic initialization. The idea of optimistic initialization is to ensure a better exploration of the state space as explained in Section 4.6. For the optimistic initialization, all values are set to 0.0 which is the largest possible q-value. In particular, the algorithm is run with these four settings:

1. reward function R^1, random initialization,
2. reward function R^2, random initialization,
3. reward function R^3, random initialization, and
4. reward function R^1, optimistic initialization.

The remaining two settings (R^2 and R^3 paired with optimistic initialization) are both omitted because the reward function and initialization method do not have very much influence on the quality of the solution. This is showed in the following. The other (hyper)parameters are set to the found values of the last chapter, listed again in Figure 7.1. As seen in Section 6.4 the secret mapping itself has a big influence on the performance of the algorithm. For this final evaluation, there are no constraints set for the secret mappings. For each of the four settings listed above the algorithm is run 30 times for the same 30 mappings. Additionally, the SMT solver (Section 2.6) is also run 30 times for the same mappings. The resulting mappings of the runs of the algorithm and the runs of the SMT solver are again evaluated using random experiments. The tabular q-learning algorithm uses experiments of lengths 2, 5, and 8 to find the secret mapping. The SMT solver on the other hand infers a mapping that matches the experiment throughputs of the secret mapping with the experiment length being lower

# ports :	8	mapping shape :	True
# instructions :	310	exp_{number} :	30
ϵ_{decay} :	~ 0.9998	exp_{length} :	$2, 5, 8$
ϵ_{end} :	0.01	mapping constraints :	1
α :	0.002	training levels :	3
γ :	0.99	exp_{distr} :	*equal*

Figure 7.1: Parameter setting for the remaining (hyper)parameters

or equal to a defined instruction bound. Therefore different experiment lengths are used to evaluate the resulting mappings, starting with an experiment length of 2.

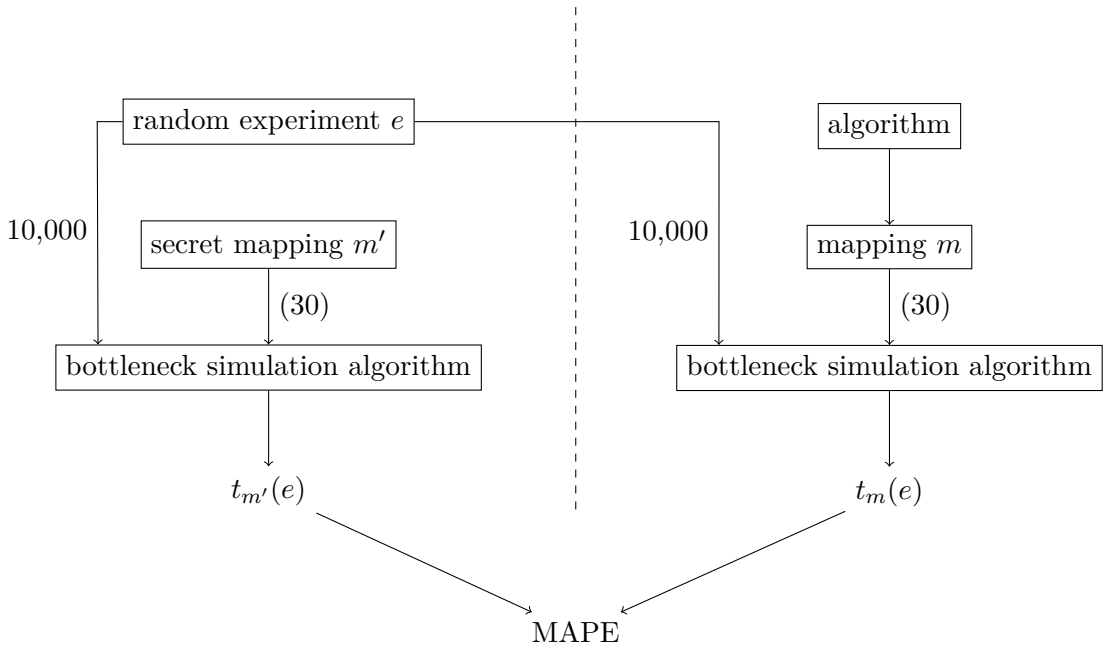


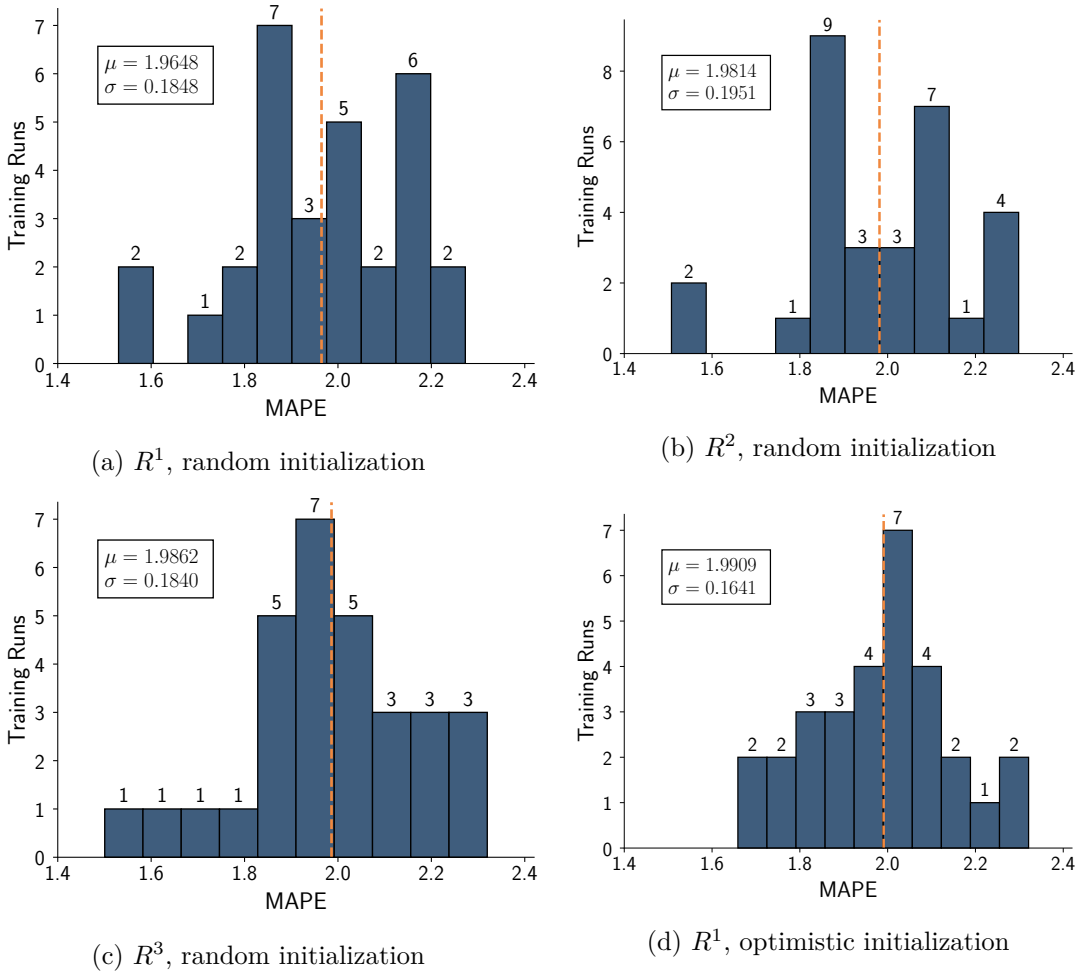
Figure 7.2: Structure of the evaluation process

The structure of the evaluation process is visualized in Figure 7.2. The left side of the diagram depicts the throughput generation of the experiment with the secret mapping. This is done when the experiment is created as described in Section 5.1.2. The right side of the diagram depicts how such an experiment is evaluated with the mapping the corresponding algorithm found. Note that which algorithm is not further specified, the right hand side is done 5 times: 4 times the reinforcement learning algorithm with the different settings listed above and additionally to that, with the SMT solver. The two throughputs are used to calculate the relative error. There are 10,000 experiments

generated, leading to 10,000 relative errors. These relative errors can then be used to calculate the MAPE. So there is one MAPE for the solution of each of the 5 algorithms for the same secret mapping. And this process is then repeated for 29 other secret mappings. Alltogether there are 30 MAPEs for each of the 5 algorithms.

7.1 Evaluating with Experiments of Length 2

As already described, the solutions are first evaluated with experiments with a maximal length of 2. The distribution of the resulting MAPEs for the reinforcement learning algorithms are gathered in Figure 7.3.



(a) R^1, random initialization

(b) R^2, random initialization

(c) R^3, random initialization

(d) R^1, optimistic initialization

Figure 7.3: Histograms showing the distribution of the MAPEs for the 4 algorithms

It can be seen that the resulting means of each of the 4 settings are very close. Due to that it cannot be concluded which of the 4 settings produces the best results, but it

seems that the setting with R^1 and random initialization of the q-table is slightly better than the other ones.
The 30 mappings the SMT solver inferred are evaluated in the same way, i.e. using 10,000 experiments. The histogram depicting the 30 MAPEs for these results can be seen in Figure 7.4. This histogram shows that the results of the SMT solver are much worse

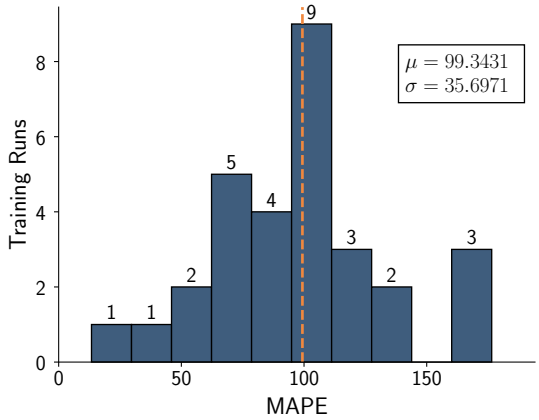
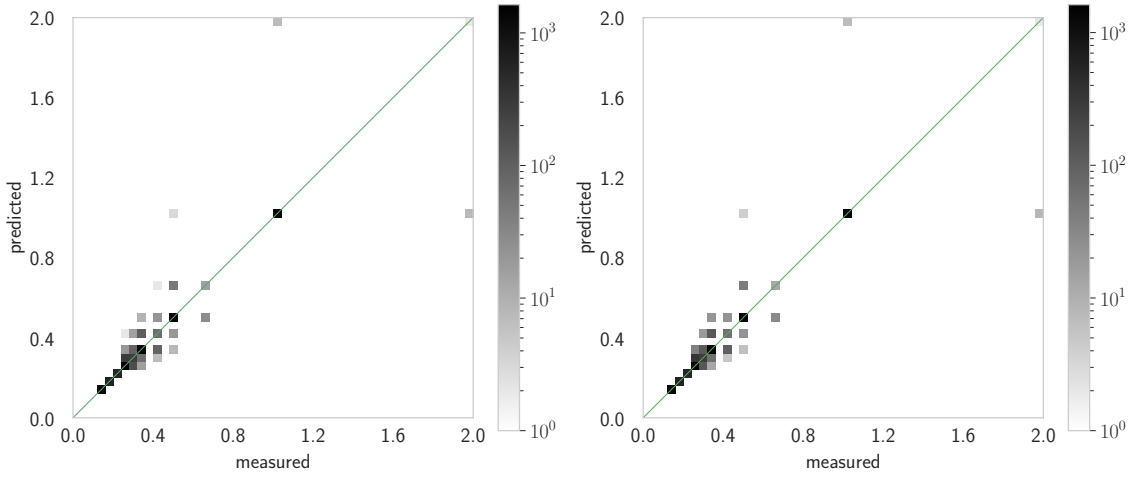


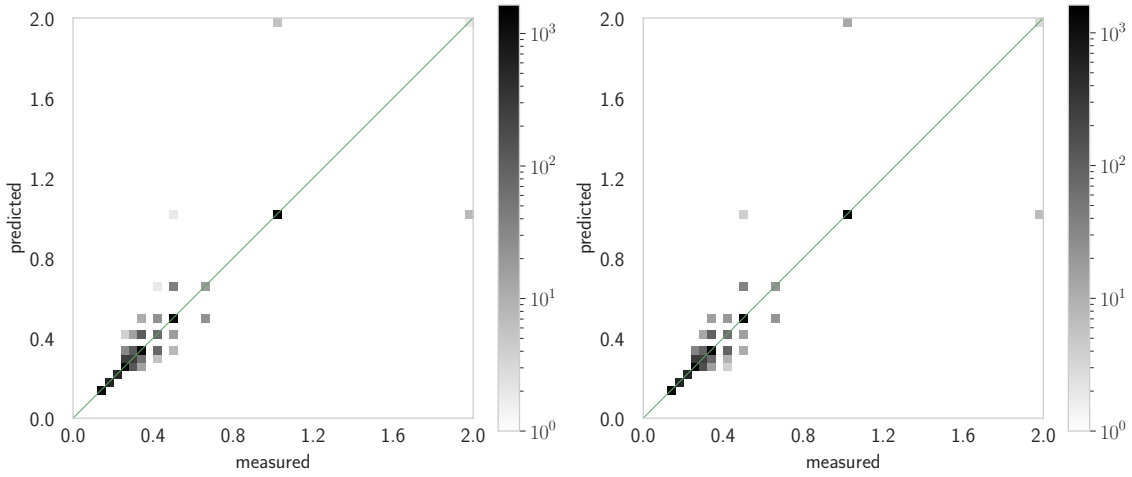
Figure 7.4: Histogram showing the distribution of the MAPEs for the SMT solver

as the mean of the MAPEs is about 50 times higher than the mean of the MAPEs of the reinforcement learning algorithms (no matter which of the four settings). This seems very unintuitive as the SMT solver should find mappings whose throughputs are very close (up to a defined bound) to the throughputs of the corresponding secret mappings (as long as it terminates within a predefined time span). To analyze this it helps to take a closer look at one of these secret mappings and how the SMT solver's solution compares to the reinforcement learning algorithms' solutions in detail. Figure 7.5 shows several heatmaps that depict the correlation of the found mapping and the secret mapping for the different runs. They all correspond to the same secret mapping. In the heatmap, a square corresponds to a number of experiments. A square with a darker grey level represents more experiments. Squares on the straight line in the middle are experiments that are predicted perfectly, meaning the throughputs of these experiments with secret and found mapping are the same. Blocks on the left side of the line mean that the throughput of the experiments using the found mapping was too large and on the right side that the throughput was too small. So the heatmap describes how well the resulting throughputs correlate. Additionally to the heatmaps for the solution of the 5 algorithms there is a sixth heatmap. This sixth heatmap is an evaluation of the 10,000 experiments with the secret mapping itself, so the perfect solution. This can be seen as the squares all lie on the line perfectly.
These heatmaps correspond to the secret mapping of the first of the 30 training runs. Displaying such heatmaps for all 30 secret mappings for different experiments lengths would take too much space, especially as the found mappings are evaluated with different experiment lengths in the following. Therefore just the ones corresponding to the first training run are shown. It can be seen that the upper 4 heatmaps are *very* similar. For some of the squares the difference in colour might be noticeable, but most of the squares



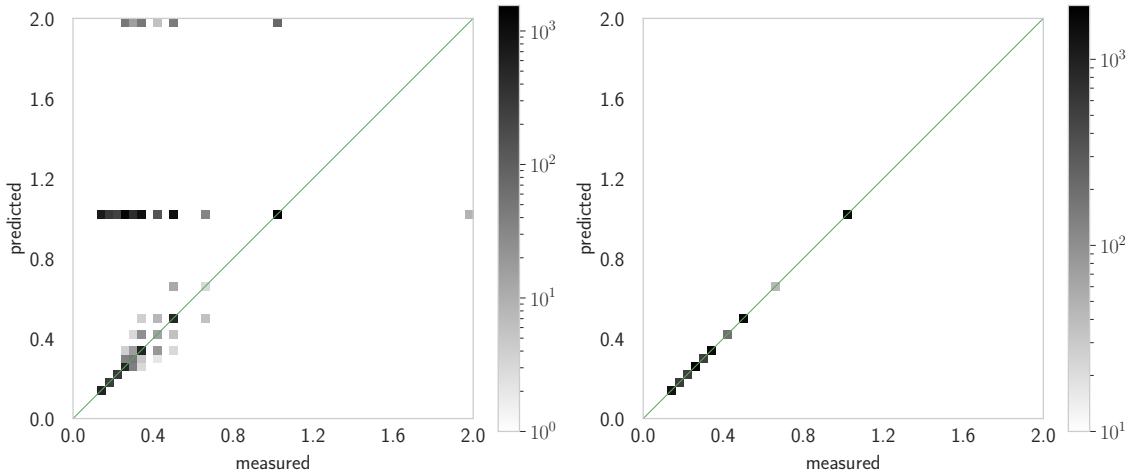
(a) R^1, random initialization

(b) R^2, random initialization



(c) R^3, random initialization

(d) R^1, optimistic initialization



(e) SMT solver

(f) perfect solution

Figure 7.5: Heatmaps visualizing the correlation of found and the secret mapping

look the same. This fits to the histograms showing the distribution of the MAPEs (Figure 7.3) as the means of these MAPEs are also very close. Nevertheless, the heatmap corresponding to the SMT solver's solution looks different. It can be seen that the predicted throughputs do not match the measured throughputs of the secret mapping for much more experiments than in the other heatmaps. That is because there are more squares further away from the correlation line and in addition to that, these squares are fairly dark, i.e. they represent a lot of experiments. This means that the SMT solver does not work as intended. If working correctly it should find a mapping that produces almost perfect results - not for any experiment, but - for experiments of a length that does not exceed the predefined instruction bound, in this case, 2. This issue seems to have something to do with the size of the secret mapping that is inferred. That is because for smaller mappings the SMT solver produces quite good solutions. Figure 7.6 shows the heatmap for a corresponding solution of the SMT solver for a smaller secret mapping. In this case, the architecture consists of 8 ports and 10 instructions, with the instruction bound being set to 5. The heatmap shows the correlation of 10,000 random experiments of length up to 5. The resulting MAPE is 0.00% and it can be seen that

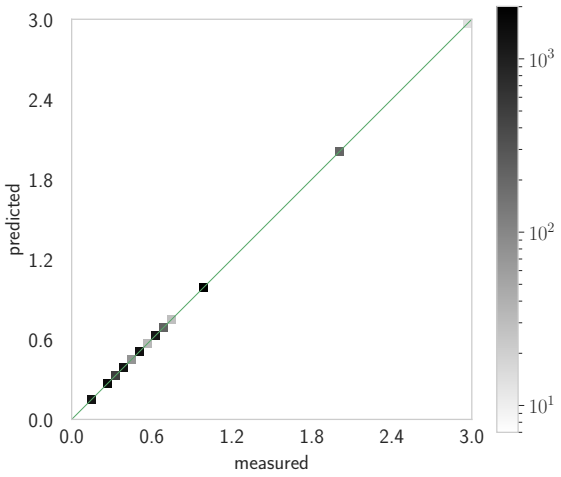
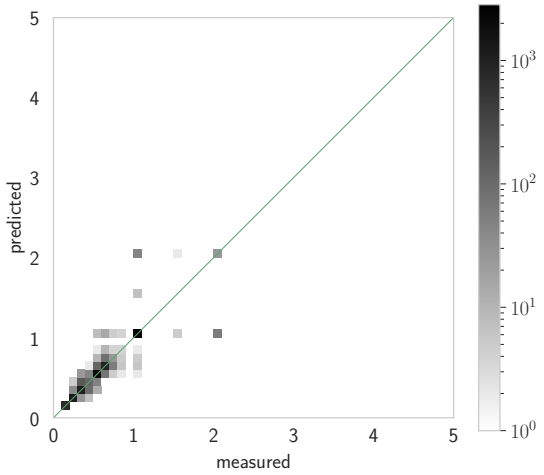


Figure 7.6: Heatmap for a solution of the SMT solver for a smaller mapping

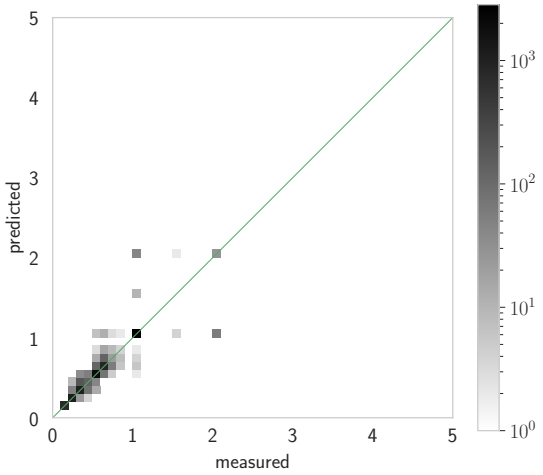
the results correlate perfectly.
Thus, we conclude that the size of the corresponding secret mappings is the reason for the SMT solver's solution not being optimal. The SMT solver seems to stop its inference process before it should, leading to this imperfect solution.

7.2 Evaluating with Experiments of Length 5

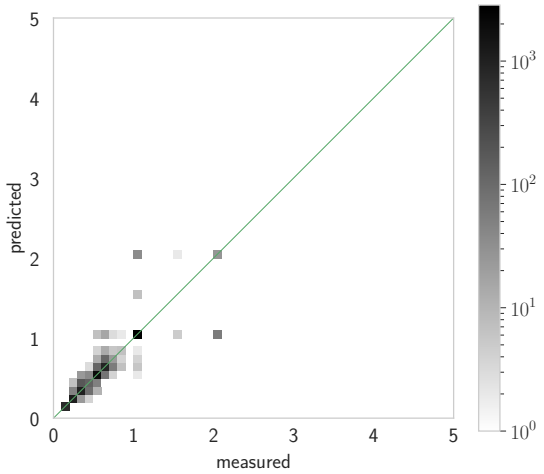
Next, the results are evaluated with experiments with a maximal length of 5. Figure 7.7 shows the heatmaps for the found solutions for the first of the 30 runs. The histograms depicting the distribution of the MAPEs is not shown as the result is not much different



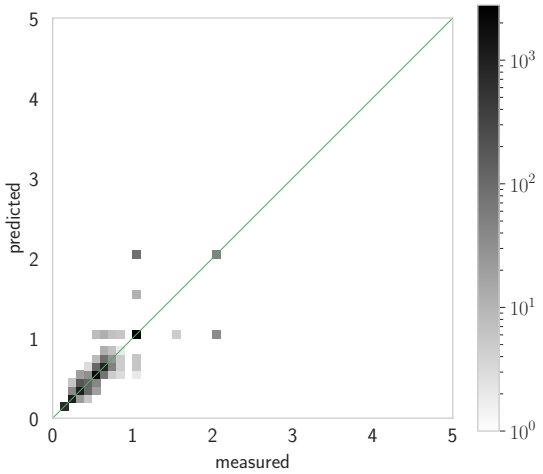
(a) R^1, random initialization



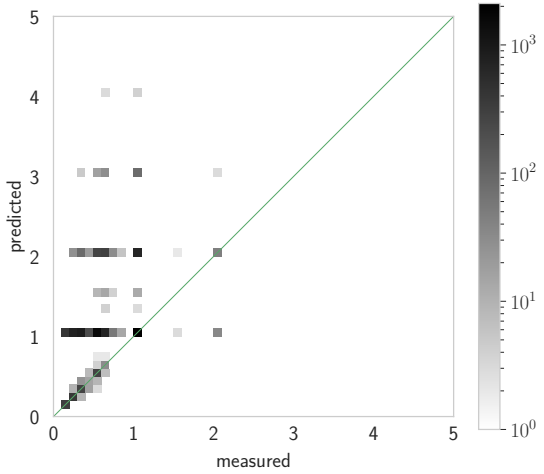
(b) R^2, random initialization



(c) R^3, random initialization



(d) R^1, optimistic initialization



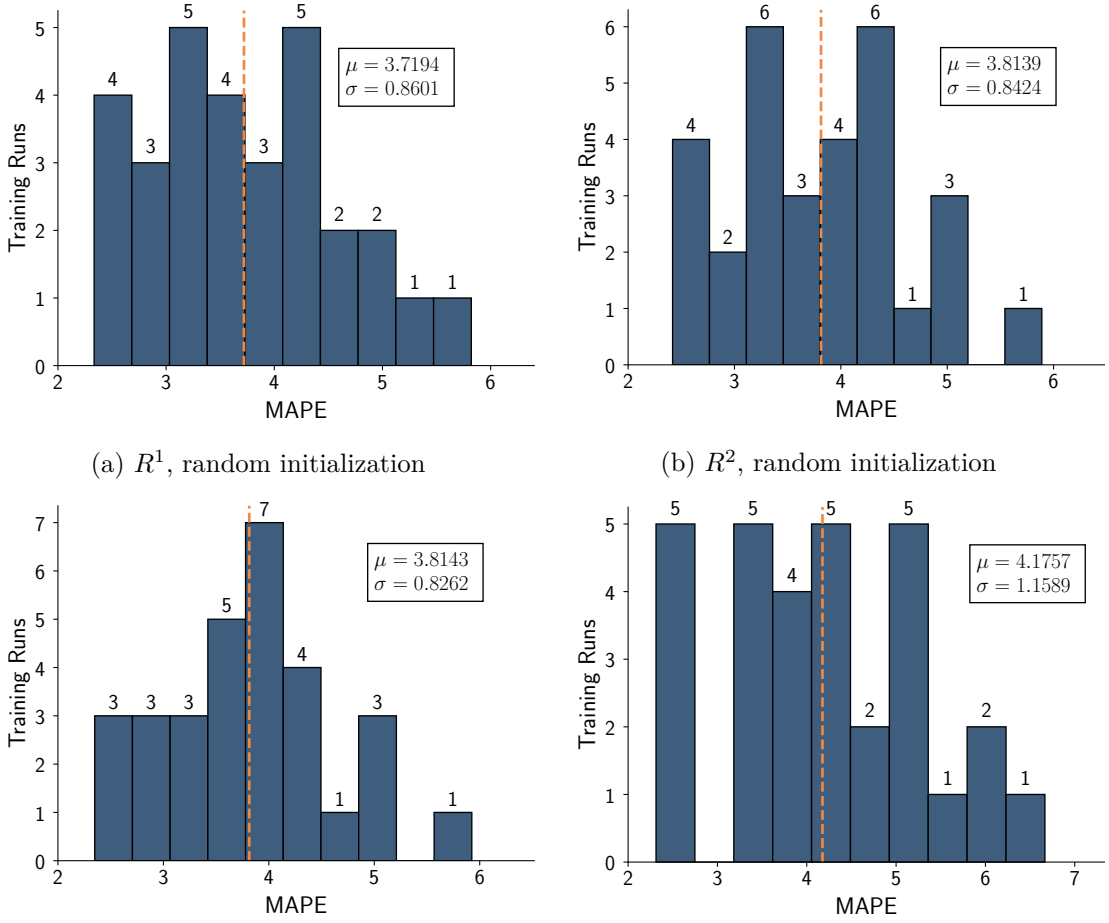
(e) SMT solver

Figure 7.7: Heatmaps visualizing the correlation of found and the secret mapping

to the distributions seen before (Figure 7.3). When increasing the instruction bound in the SMT solver it does not find better mappings. The corresponding heatmap visualizes that.

7.3 Evaluating with Experiments of Length 20

Next, the results are evaluated with 10,000 experiments with a maximal length of 20 to see how the results change for longer instruction sequences. Figure 7.8 shows the distribution of the MAPEs for each of the 4 versions of the algorithm.



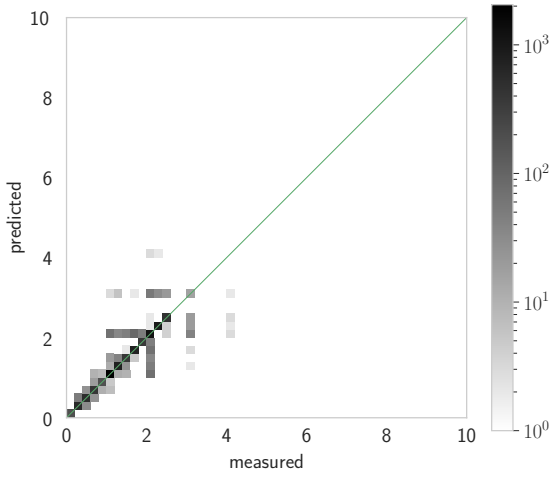
(a) R^1, random initialization

(b) R^2, random initialization

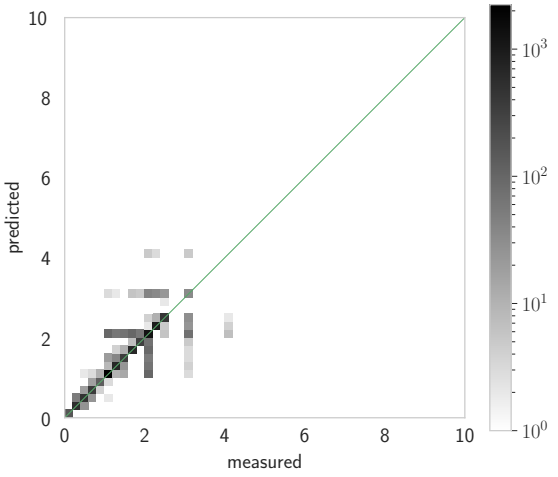
(c) R^3, random initialization

(d) R^1, optimistic initialization

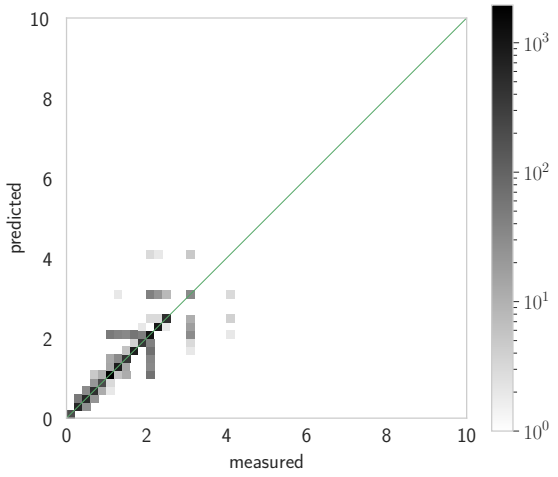
Figure 7.8: Histograms showing the distribution of the MAPEs for the four versions



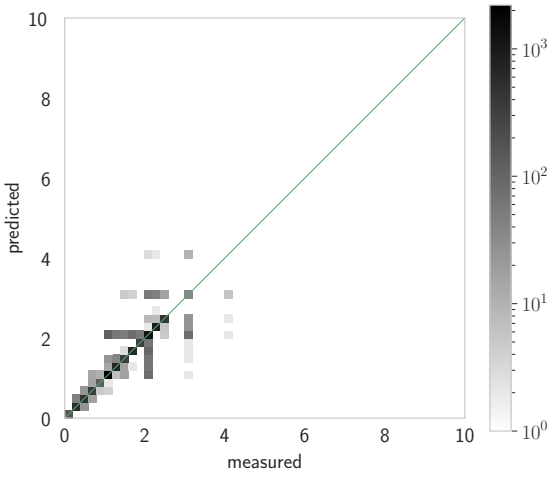
(a) R^1, random initialization



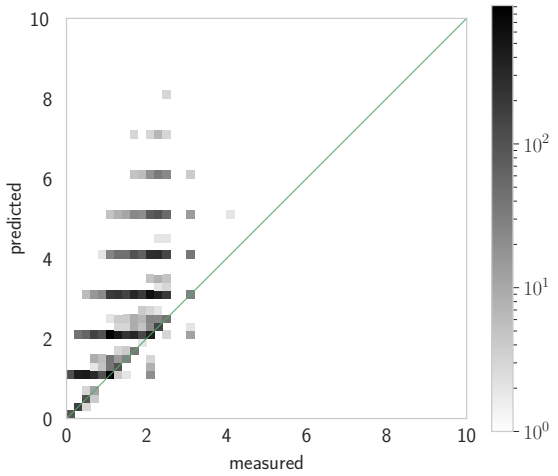
(b) R^2, random initialization



(c) R^3, random initialization



(d) R^1, optimistic initialization



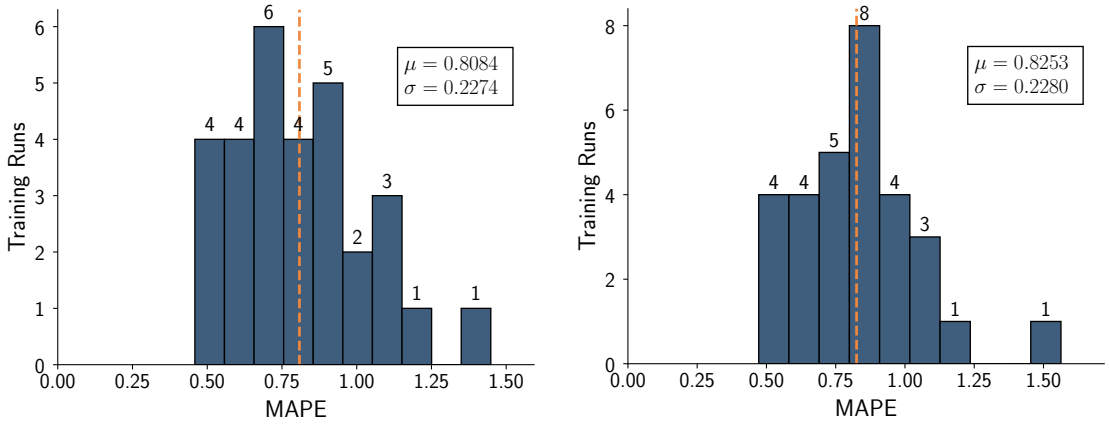
(e) SMT solver

Figure 7.9: Heatmaps visualizing the correlation of found and the secret mapping

The histograms show that the results lie in the same range again, but this time there is more difference noticeable between the four settings. The setting with R^1 and random initialization seems to be the best setting with a mean MAPE of 3.7. The two other versions with random initialization are about equal. The variant that uses optimistic initialization is clearly the worst as the corresponding mean MAPE is about 0.3 greater than the one of the two other versions, which is 10% worse in total. In addition to that, the standard deviation is also worse than for the other 3 settings. As explained in Section 4.6, optimistic initialization of the q-table results in more exploration. The state space in the q-table is reduced (as described in Section 5.4.1) and therefore the trajectories in the MDP are shorter. Due to that, optimistic initial values do not necessarily have the effect that every possible state of the original MDP is explored. That is the reason why optimistic initial values do not improve the quality of the port mapping the agents find.
That the MAPE for the solutions is still around 4% can also be seen in the heatmaps. Figure 7.9 depicts the heatmaps for the first mapping (as in the last sections) for 10,000 random experiments with a maximal length of 20. The heatmaps show that the found mapping produces similar throughputs as the secret mapping, with stronger discrepancies for integer throughputs like 2, 3, and 4. It is hard to see by looking at these heatmaps where the four settings produce different results, the most noticeable changes are for the predicted and measured throughputs of 4 and 3 (the outliers). Apart from that, they are quite similar again. The mapping by the SMT solver produces results that have much greater throughputs than measured for the secret mapping.

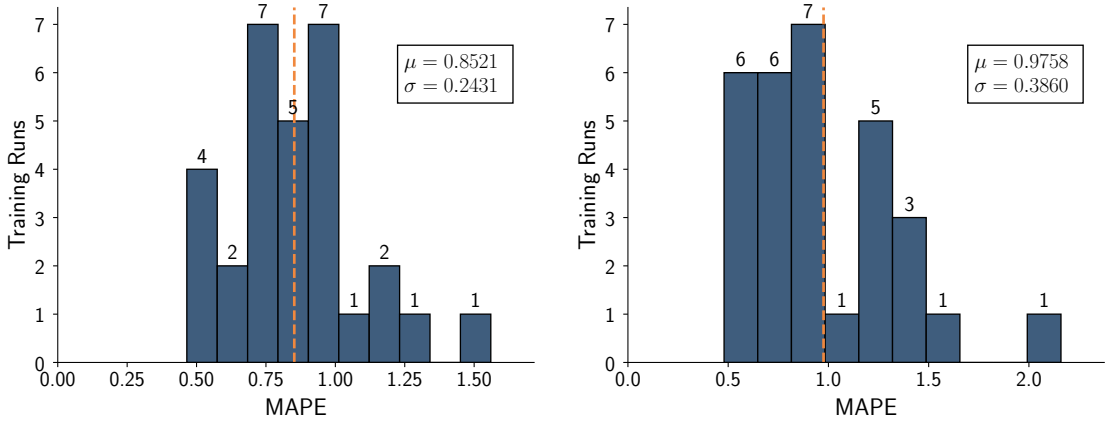
7.4 Evaluating with Experiments of Length 100

Eventually, the quality of the found mappings is evaluated with much larger experiments. This is particularly interesting as the algorithm only uses experiments of a maximal length of 8 during training to find the secret mapping. In the following, the found mappings are evaluated by 10,000 random experiments with a maximal length of 100. This shows if the algorithm's results are capable of scaling up. Figure 7.10 shows the 4 histograms with the distribution of the corresponding MAPEs. The histograms show that the resulting mapping predicts the throughputs very good with the mean of the MAPE being under 1%. Since the results are better for longer experiments the found mappings perform worse for shorter experiments. One intuitive explanation for this could be that for shorter experiments the correct mapping for each of the contained instructions weighs more. As the experiments get longer the false predictions for a few instructions do not carry as much weight anymore. The order from best to worst setting is the same as before, the first setting (with R^1 and random initialization) finds the best solutions.
Figure 7.11 shows the heatmaps for the prediction quality of the solution of the first out of the 30 runs (as in previous heatmaps). The heatmaps confirm that longer experiments are predicted better than shorter experiments. From a certain point on, the agents' mapping predicts the experiments very good. It can be seen in the histograms that for



(a) R^1, random initialization

(b) R^2, random initialization

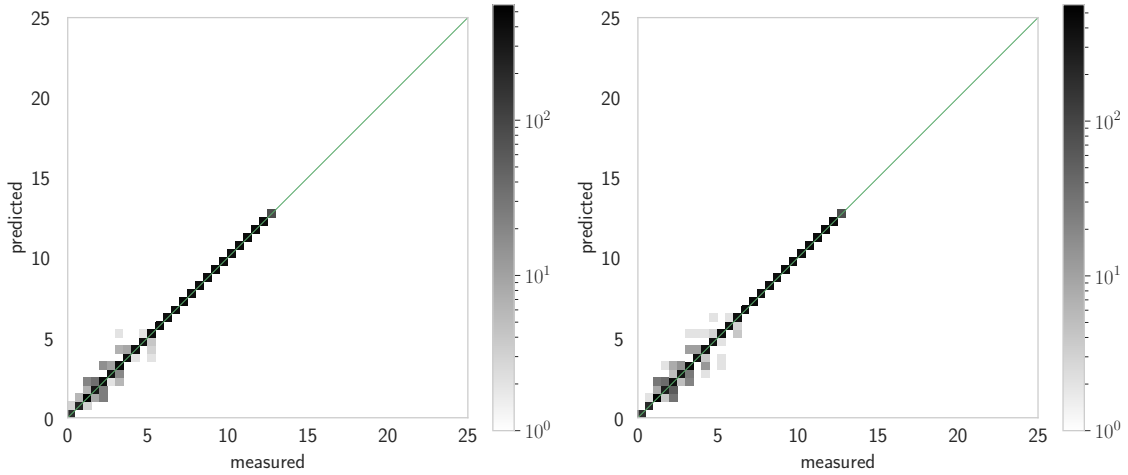


(c) R^3, random initialization

(d) R^1, optimistic initialization

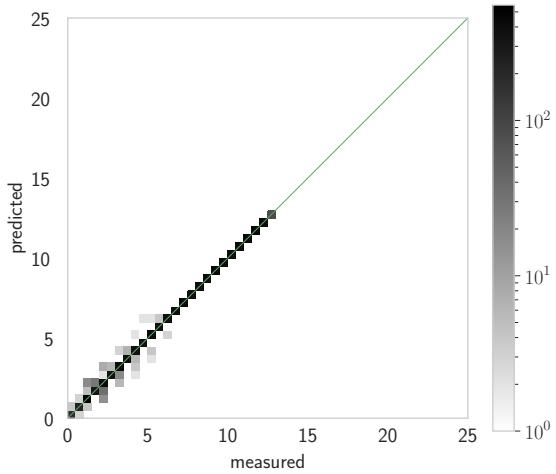
Figure 7.10: Histograms showing the distribution of the MAPEs for the four versions

this mapping this point is for experiments generating a throughput of about 8. What could be seen for the heatmap of the mapping the SMT solver found is visible more extremely for these larger experiments.

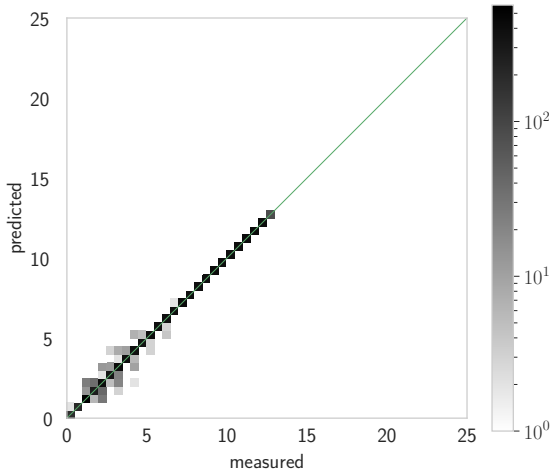


(a) R^1, random initialization

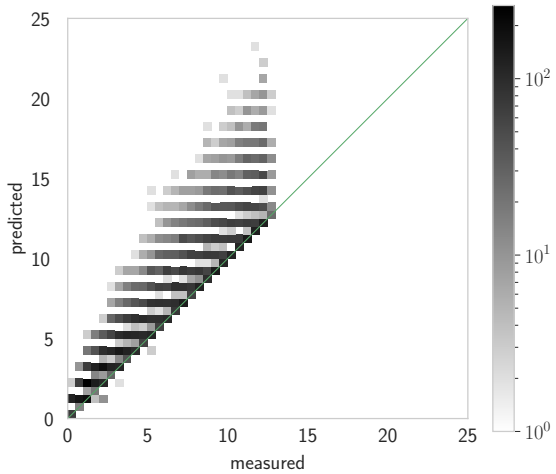
(b) R^2, random initialization



(c) R^3, random initialization



(d) R^1, optimistic initialization



(e) SMT solver

Figure 7.11: Heatmaps visualizing the correlation of found and the secret mapping

8 Conclusion and Future Work

In this thesis, we developed the first reinforcement learning algorithm that is able to infer the port mapping of a given processor in the two-level model by only having information about the throughput for randomly generated experiments. The algorithm uses tabular q-learning for several agents in combination with curriculum learning with 3 levels of training, increasing the maximum experiment length from level to level. We developed an alternate version of the algorithm that infers the shape of the secret mapping before starting to train the agents and uses this information to act accordingly. We saw that this alternate version produces much better results, but it leads to the agents having such a high quality in the beginning of a training level that there is no steady learning progress for the rest of this level. The algorithm has a lot of different hyperparameters and parameters that can be adjusted. We did a partial grid search to find good values for the hyperparameters. Afterwards, we evaluated the resulting algorithm with 4 different settings for the remaining parameters and compared the results to those of an SMT solver. This evaluation showed that the SMT solver cannot compete with the reinforcement learning algorithms, no matter which of the 4 settings. The evaluation showed that the algorithm is capable of finding a solution that produces similar throughputs for given experiments. The mean average error of the throughputs of the found mapping and the secret mapping is under 5% for a given set of experiments. Despite these few errors the found mapping still resembles the behaviour of the secret mapping and can therefore be used as an equivalent.
It is outstanding that, although only using small experiments during training, the designed algorithm also performs very good for large experiments. To be more specific, the resulting mapping the agents find performs even better for larger experiments than for smaller ones. As existing work includes the decomposition of instructions into their μops it cannot be used to compare the results to those of our algorithm.
Since the algorithm produces already good results and some of the hyperparameters were not analyzed due to the limited scope of this thesis, it might be worth doing that in the future as it could improve the solution quality even more. As already explained in Chapter 6 it would be interesting to analyze how different values of the learning rate α, the random exploration coefficient ϵ, and the number of episodes have an influence on the solution quality of the algorithm. Further, as the equal experiment distribution leads to better results it could also be investigated if the solution quality is improved even more when using an unequal experiment distribution where longer experiments have a higher probability to be chosen.
Since this is the first approach of using reinforcement learning to infer the port mapping of a given processor and it works quite well, other reinforcement learning strategies could be tried out to solve this problem in the future. It could be seen that the tabular q-learning algorithm developed in this work is not able to improve its solution from an early point on. Therefore it would be interesting to see if a deep Q-network (DQN) [19] is able to perform better with regards to this issue and provide a better solution in

general. It has been shown that DQN agents are often able to find good solutions to complex problems as they have the ability to generalize. As for mappings of large size, the problem becomes really complex and using DQN agents could therefore potentially give better results. On the one hand, the DQN agents could be built as a new algorithm to see if a DQN works better. On the other hand, it could be interesting to see if a DQN can make use of structures or results of the algorithm implemented in this work (for example by inferring the mapping shape in advance the reward for chosen actions by the DQN agents can be adjusted correspondingly if the number of chosen actions fits or does not fit to the shape). Barret et al. [3] also train a DQN in their work to solve a combinatorial optimization problem on a graph. They run their algorithm 50 times on the same graph with different random initializations. As the initial values can have a big impact on the solution quality this improves the outcome. The best out of the 50 solutions can be chosen. The same principle can also be applied to the algorithm presented in this work: by running the algorithm 50 times for the same mapping with different random initialization of the q-table each run there are 50 resulting mappings that can be compared and the best one can be chosen. This strategy could also be applied if a DQN agent is implemented to find the secret mapping.
Real-world processors decompose each instruction into its μops and then execute these μops on the ports instead of the instructions themselves. Port mappings depicting this process are denoted as three-level mappings (as described in Chapter 2). The algorithm developed in this work only works for mappings in the two-level model. However, it can be used for future work solving the problem in the three-level model. As the instructions are not further specified in this work they could also be seen as μops. Looking at it that way the algorithm solves to find the second half of a mapping in the three-level model, i.e. the mapping of all μops to the corresponding ports that can execute these μops. So when using the algorithm for this part there only has to be developed an algorithm that finds the first half of the mapping, i.e. the decomposition of instructions into their μops.

Bibliography

[1] Andreas Abel and Jan Reineke. uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 673–686, 2019.

[2] Yunsheng Bai, Derek Xu, Alex Wang, Ken Gu, Xueqing Wu, Agustin Marinovic, Christopher Ro, Yizhou Sun, and Wei Wang. Fast detection of maximum common subgraph via deep q-learning. *arXiv preprint arXiv:2002.03129*, 2020.

[3] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3243–3250, 2020.

[4] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1553374.1553380.

[5] Andrea Di Biagio. llvm-mca: a static performance analysis tool, 2018. URL: http://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html.

[6] Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

[7] Guillaume Chatelet. llvm-exegesis: Automatic measurement of instruction latency/uops, 2018. URL: https://lists.llvm.org/pipermail/llvm-dev/2018-March/121814.html.

[8] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In *Brazilian Symposium on Formal Methods*, pages 23–36. Springer, 2009.

[9] Franklin Cardeñoso Fernandez and Wouter Caarls. Parameters tuning and optimization for reinforcement learning algorithms using evolutionary computing. In *2018 International Conference on Information Systems and Computer Science (INCISCOS)*, pages 301–305. IEEE, 2018.

[10] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and

micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.

[11] Google. Exegesis: Automatic measurement of instruction latency/uops, 2018. URL: `https://github.com/google/EXEgesis`.

[12] Intel. Intel architecture code analyzer, [n.d.]. URL: `https://software.intel.com/content/www/us/en/develop/articles/intel-architecture-code-analyzer.html`.

[13] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30:6348–6358, 2017.

[14] Kai A. Krueger and Peter Dayan. Flexible shaping: How learning in small steps helps. *Cognition*, 110(3):380 – 394, 2009. URL: `http://www.sciencedirect.com/science/article/pii/S0010027708002850`, `doi:https://doi.org/10.1016/j.cognition.2008.11.014`.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[16] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131. IEEE, 2018.

[17] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey, 2020. `arXiv:2003.03600`.

[18] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pages 4505–4515. PMLR, 2019.

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

[20] Sanmit Narvekar and Peter Stone. Learning curriculum policies for reinforcement learning. *arXiv preprint arXiv:1812.00285*, 2018.

[21] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multi-agent systems: A review of challenges, solutions and applications. *CoRR*, abs/1812.11794, 2018. URL: `http://arxiv.org/abs/1812.11794`, `arXiv:1812.11794`.

[22] Martijn Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. *Reinforcement Learning: State of the Art*, pages 3–42, 01 2012. `doi:10.1007/978-3-642-27645-3_1`.

[23] Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–455. IEEE, 2020.

[24] Fabian Ritter and Sebastian Hack. PMEvo: Portable inference of port mappings for out-of-order processors by evolutionary optimization. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 608–622. ACM, 2020. `doi:10.1145/3385412.3385995`.

[25] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[26] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In *Advances in neural information processing systems*, pages 385–392, 1995.

[27] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, January 1967. `doi:10.1147/rd.111.0025`.

[28] Mohan Yogeswaran and SG Ponnambalam. Reinforcement learning: exploration–exploitation dilemma in multi-agent foraging task. *Opsearch*, 49(3):223–236, 2012.

[29] Elena Zaharieva-Stoyanova and Lorentz Jäntschi. Detection of software data dependency in superscalar computer architecture execution. In *Proceedings of the 4th international conference conference on Computer systems and technologies: e-Learning*, pages 107–112. Citeseer, 2003.