Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Project thesis (Computer Science)

Reinforcement Learning in a Multi-Agent System for Train Scheduling

Authors	Ralph Meier Dano Roost
Main supervisor	Andreas Weiler
Sub supervisor	Thilo Stadelmann

Date	20.12.2019

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum: Unterschriften:

…… ……

……..

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Abstract bzw. dem Management Summary mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

Die steigende Anzahl der Pendler bring die Kapazität des Schienennetzes der Schweizerischen Bundesbahnen SBB immer mehr an seine Grenzen. Da der Ausbau der Infrastruktur nicht mit den Passagierzahlen mithalten kann, bemüht sich die SBB, mehr Züge in dichteren Abständen auf ihr Netz zu bringen. Für die Planung des dichteren Zugverkehrs experimentiert die SBB auch mit Technologien des maschinellen Lernens, insbesondere mit Reinforcement Learning (RL). Dabei soll ein Algorithmus selbstständig Strategien erlernen, um das gegebene Problem zu lösen. In diesem Fall handelt es sich dabei um das Steuern von Zügen, welche möglichst zeitnah zu ihrem Ziel geführt werden sollen. Kollisionen gilt es dabei zu verhindern und potenzielle Hindernisse wie defekte Züge müssen umfahren werden. Auch können Züge mit verschiedenen Geschwindigkeiten unterwegs sein, was bei der Planung berücksichtigt werden soll.
Diese Arbeit ist dabei ein Beitrag zur sogenannten Flatland Challenge, einem Wettbewerb der SBB und der Crowdsourcing-Plattform AICrowd, bei welchem das beschriebene Problem mittels einer zur Verfügung gestellten Simulation des Schienennetzes gelöst werden soll. Der Wettbewerb ist aufgeteilt in zwei Runden mit wachsendem Schwierigkeitslevel. Diese Schienensimulation erlaubt es, eigene Lösungen für das Problem zu trainieren und zu evaluieren. Die Schwierigkeit dabei ist, die Züge so zu steuern, dass sie sich auch in komplexen Situationen nicht blockieren. Dazu ist primär die Zusammenarbeit der Züge von grosser Bedeutung.
Der präsentierte Lösungsansatz verwendet den Asynchronous Advantage Actor-Critic Algorithmus, einen der derzeit besten RL-Algorithmen, welcher verteiltes Lernen ermöglicht. Der präsentierte Ansatz orientiert sich dabei stark an der Ausgangslösung von Stephan Huschauer. Nach der ersten lauffähigen Version wurde der Ansatz nach und nach mit unterschiedlichen Features erweitert, wie beispielsweise einem veränderten Kontrollmechanismus der Züge, Curriculum Learning oder verteiltem Lernen über mehrere Rechner. Um den Fortschritt zu quantifizieren wurden für sämtliche Änderungen Experimente durchgeführt. Durch das Hinzufügen unterschiedlicher Features konnte eine signifikante Verbesserung der Performanz verzeichnet werden. In Runde 1 war es dabei möglich, die Ankunftsrate von den 16.6 % der Ausgangslösung auf 48.9 % zu verbessern. Für die anspruchsvolleren Eisenbahnnetze der 2. Runde, für welche keine Ausgangslösung existiert, konnte dank weiteren Verbesserungen ebenfalls ein respektables Resultat erzielt werden. Die präsentierte Lösung konnte eine Ankunftsrate von 29.1 % erreichen, was zum Evaluierungszeitpunkt dem 4. Platz von insgesamt 24 teilnehmenden Teams entsprach. Für einen praktischen Einsatz ist die präsentierte Lösung jedoch noch nicht geeignet, dies bedarf weiterer Forschung.

Abstract

The increasing number of commuters is pushing the capacity of the Swiss Federal Railways SBB rail network to its limits. Since the expansion of the infrastructure cannot keep up with the increasing number of passengers, SBB is exploring new ways to bring more trains onto its network. To achieve this goal, SBB is also experimenting with machine learning technologies such as reinforcement learning (RL). The goal for this type of algorithm is to independently learn strategies in order to solve the problem at hand, in our case the challenge of guiding trains to their assigned destinations. Besides not colliding with other trains, it is also necessary to avoid potential obstacles such as defective trains and to take different speed profiles into account.
This work is a contribution to the Flatland Challenge, a competition published by SBB and the crowdsourcing platform AICrowd, which aims to solve the problem described by using a provided simulation of the rail network. The competition is divided into two rounds with increasing difficulty. This rail simulation allows us to train and evaluate own solutions for the given problem. The main task is to control the trains in cooperative way so they do not block each other even in complex situations. The selected solution uses the Asynchronous Advantage Actor-Critic Algorithm, a state of the art RL algorithm which supports distributed learning. The approach presented is based on the baseline solution presented by Stephan Huschauer. After establishing the first running version, the approach was improved with an array of new features such as a modified train control mechanism, curriculum learning or distributed learning over multiple computers. To quantify the progress, experiments were conducted for all applied changes. By adding these features, a significant improvement could be achieved. In round one, it was possible to improve the train arrival rate from 16.6% of the baseline solution to 48.9%. For the more demanding second round, there is no baseline available. The solution presented was able to achieve a train arrival rate of 29.1%, which corresponded at submission time to the 4th place of a total of 24 participating teams. However, the solution presented is not yet suitable for practical use and requires further research to achieve a useful performance in a real-world scenario.

Preface

We would like to give special thanks to:

- Andreas Weiler and Thilo Stadelmann for their great support during this work, for their helpful tips and for pushing us in the right direction.
  We are grateful for the opportunity to dive deep into the field of reinforcement learning as part of this project.
- Remo Maurer for being very generous with providing computing infrastructure, especially the infrastructure test server.

Contents

1 Introduction

1.1 Baseline

This work explores a real-world usage of multi-agent reinforcement learning (RL) for controlling train traffic in a complex railway system. As part of the Flatland challenge, a contest created by the Swiss Federal Railways SBB and the crowdsourcing platform AICrowd [1], we try to improve the performance of RL based train guidance and rescheduling. The goal of the challenge is to successfully guide all trains to their assigned target stations in a simulated environment called Flatland environment. This is challenging because a single wrong decision can cause a chain reaction that makes it impossible for many other trains to successfully reach their destinations. The endeavor is further complicated by trains with different speed profiles and the possibility of malfunctioning trains. In the words of SBB and AICrowd, the challenge is described as follows [1]:

> The Flatland Challenge is a competition to foster progress in multi-agent reinforcement learning for any re-scheduling problem (RSP). The challenge addresses a real-world problem faced by many transportations and logistics companies around the world (such as the Swiss Federal Railways, SBB). Different tasks related to RSP on a simplified 2D multi-agent railway simulation must be solved. Your contribution may shape the way modern traffic management systems (TMS) are implemented not only in railway but also in other areas of transportation and logistics. This will be the first of a series of challenges related to re-scheduling and complex transportation systems.

The challenge consists of two parts [1].

- Part 1 includes avoiding conflicts with multiple trains (agents) on a given environment. The difficulty thereby is, that the layout of the environment is not known upfront.
- Part 2 aims to optimize train traffic which includes trains with different speed profiles, malfunctioning trains, fewer switchover facilities and in general more scheduled trains in a shorter time.

This work is based on the work of Stephan Huschauer [2] and further investigates the idea to use the Asynchronous Advantage Actor-Critic Algorithm (A3C) [3], a state of the art RL algorithm, to solve the task. Besides the work of S. Huschauer, this work is also related to the work of Bacchiani, Molinari and Patander [4]. Their work also aims to apply the A3C algorithm in a cooperative multi-agent environment and investigates communication free cooperation. Unlike the Flatland challenge, the goal of this work is to cooperate on a road traffic environment. By applying the A3C algorithm, the work shows that it is possible to learn cooperation by treating the other agents as part of the environment. Both the works of Bacchiani, Molinari and Patander as well as the work of S. Huschauer use a shared policy for all acting agents.

1.2 Goal of this work

The aim of the work is to explore the use of the A3C algorithm in the Flatland multi-agent environment and to improve on the approach of S. Huschauer [2].
While there may be better ways to solve the given problem than reinforcement learning, we mainly focus on pure RL but give our intuition in chapter 5 on how the explored approach could work together with other techniques to improve its success. This work is targeted towards an audience with a brief understanding of deep reinforcement learning. A basic introduction to the topic is given in section 2.1. This introduction is focused on the techniques required to understand the applied A3C algorithm and does not cover the whole field of RL. Also, an introduction into the Flatland environment can be found in section 2.2. For a deeper understanding of the complex Flatland system, it is recommended to study the Flatland documentation and specification [5] as well as the official Flatland introduction [1].

1.3 Work Approach and Sectioning

We divide this work into 4 main sections:

- **Basic Implementation:** An overview of our basic implementation to solve the Flatland challenge. Also, we shortly describe the technologies used.
- **Experiment Design and Analysis:** We identify parts of the implementation that offer room for improvement. To verify our work, we create experiments and analyse them afterward.
- **Results:** A discussion of the final solutions that were submitted for evaluation in the Flatland challenge for both round 1 and round 2.
- **Discussion and Outlook:** An analysis of components in the solution that would need further improvement.

We take the idea of using the A3C algorithm to solve the Flatland problem and try various modifications in an attempt to improve its performance. We proceed by giving an idea, what we want to achieve, followed by an experiment setup and an experiment analysis to either prove or disprove our hypothesis. We do this in an iterative manner, to gradually come closer to a well-performing solution for Flatland.
In our experiments, we focus exclusively on Flatland round 2. While we compare our solution for round 1 with the baseline from S. Huschauer in chapter 5, all technical improvements have also become part of our solution for round 2.

2 Technical and Theoretical Foundation

2.1 Reinforcement Learning

Basic Definitions

In recent years, major progress has been achieved in the field of reinforcement learning [6–8]. In RL, an agent learns to perform a task by interacting with an environment $\mathcal{E}$ as displayed in Figure 2.1. On every timestep t the agent a needs to take an action u. The selection of this action u is based on the current observation s. The success of the agent is measured by reward $\mathcal{R}$ received. If the agent does well, it receives a positive reward from the environment, if it does something bad, there is no or negative reward.

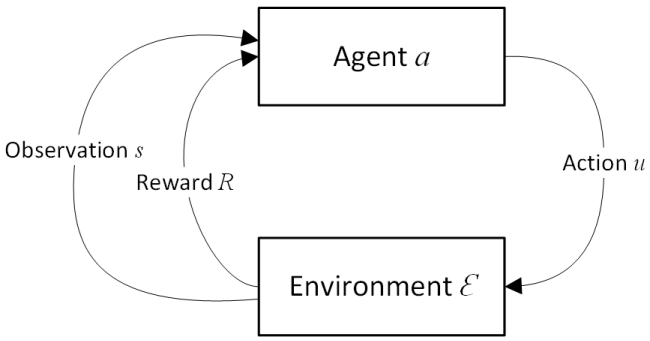


Figure 2.1: Reinforcement learning overview

The goal of the agent is now to take an action that maximizes the expected reward for all future timesteps $\mathbb{E}[\mathcal{R}_{t+1} + \mathcal{R}_{t+2} + \mathcal{R}_{t+3} + ...|s_t]$ given the current observation s_t.
This estimation should be as close as possible to the sum of actually received rewards. Often, these received rewards are discounted with a constant factor γ to the power of timestep t. With γ being something slightly less than 1, this accounts for the fact that rewards far in the future are hard to estimate. The current observation s_t, also known as the current state is used to determine which action u to take next. An agent can observe its environment either fully or partially. The cycle of taking actions and receiving a new state is repeated until the environment has reached a terminal state. Each roll-out of such an environment is called an episode.

Value-Based vs. Policy Gradient-Based Methods

Reinforcement learning methods are categorized into value-based methods and policy gradient-based methods [9, 10]. Those variants differ on how they select an action u from a given state s.

Value-based RL algorithms work by learning a value function $\mathcal{V}(s)$ through repeated rollouts of the environment. $\mathcal{V}(s)$ aims to estimate the future expected reward for any given state s as precisely as possible. Using this approximation $\mathcal{V}(s)$ we can now select the action u that takes the agent into the next state s_{t+1} with the highest expected future reward. This estimation $\mathcal{V}(s)$ is achieved by either a lookup table for all possible states or a function approximator. In this work, we solely focus on the case that $\mathcal{V}(s)$ is implemented in the form of a neural network as a function approximator. To train the neural network, we try to minimize the squared difference between the estimated reward $\mathcal{V}(s)$ and the actual reward $\mathcal{R}$:

$$loss_{value} = (\mathcal{R} - \mathcal{V}(s))^2$$

In some value-based algorithms such as Deep Q-Networks (DQN) [6], a $\mathcal{Q}(s,u)$-function is used. This function tries to estimate the expected future reward on taking action u from the given state s. We do not consider algorithms that use a $\mathcal{Q}$-function in this work.
The second category of reinforcement learning algorithms is the so-called policy gradient-based methods. These methods aim to acquire a stochastic policy $\pi(s)$ that maximizes the expected future reward $\mathcal{R}$ by taking actions with certain probabilities. Taking actions based on probabilities solves an important issue of value-based methods, which is, that by taking greedy actions with respect to state s, the agent might not explore the whole state space and misses out on better ways to act in the environment.

Asynchronous Advantage Actor-Critic Algorithm

The progress in RL has led to algorithms that combine value-based and policy gradient-based methods, generally known as actor-critic algorithms. The Asynchronous Advantage Actor-Critic Algorithm, developed by Mnih et al. [3] fits into this category. It uses both a policy $\pi(s)$ and a value function $\mathcal{V}(s)$. Both are usually separate function approximators (neural networks in our case).

- The **actor** can be seen as the policy $\pi(s)$, that selects the action u based on state s.
- The **critic** is the value function $\mathcal{V}(s)$ that estimates, how much reward can be expected from a certain state s on.

To enhance the process of learning policy $\pi(s)$, the policy loss gets multiplied by the difference between actually received reward $\mathcal{R}$ and the estimated future reward $\mathcal{V}(s)$.
This difference is called the advantage $\mathcal{A}$.

$$\mathcal{A} = \mathcal{R} - \mathcal{V}(s)$$

This advantage is then used to update the policy.

$$loss_{policy} = -\log(\pi(s_t)) \cdot \mathcal{A}$$

For actions where the received reward $\mathcal{R}$ exceeds the expected reward $\mathcal{V}(s)$ the policy update gets multiplied by a positive advantage. Therefore, the update of the neural network gets adjusted into a direction that favors experienced actions.
What makes A3C different from other actor-critic algorithms is, that it can be used in a distributed way. Many workers work at the same time on a centralized model. How we take advantage of these features is discussed in 4.3 Distributed Architecture and Parallelism.

2.2 The Flatland Rail Environment

The Flatland environment is a virtual simulation environment provided by the Swiss Federal Railway SBB and the crowdsourcing platform AICrowd. The goal of this environment is to act as a simplified simulation of real train traffic. The current state of the simulation is shown in Figure 2.2.

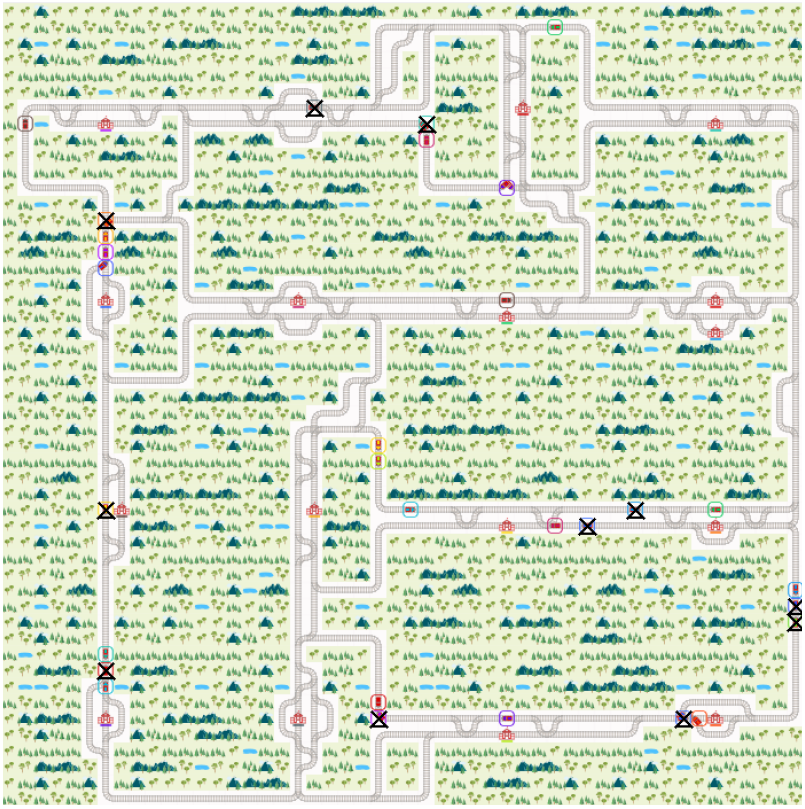


Figure 2.2: Screenshot from a running Flatland environment.

Using Flatland, we can train RL algorithms to control the actions of trains, based on observations on the grid. Flatland has a discrete structure in both its positions and its timesteps. The whole rail grid is composed of squares that can have connections to neighboring squares. In certain squares, the rail splits into two rails. On those switches, the agent has to make a decision on which action it wants to take. Dependent on the type of switch, there are different actions available as visible in Figure 2.3.

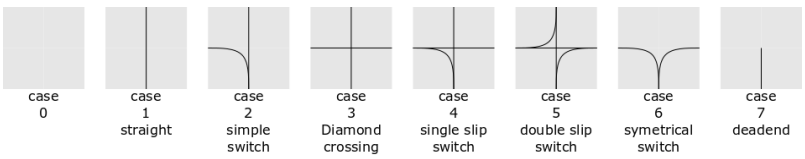


Figure 2.3: Possible switches in the Flatland environment from [5].

An exception poses switches that are approached from a side that does not allow to take an action, e.g. approaching a *case 2* switch from the top. All rail parts, independent

whether it is a switch or not also allow to take the actions to do nothing (remain halted, or keep riding), to go forward or to brake. The action space is therefore defined by:

$$U = \{\text{do nothing, go left, go forward, go right, brake}\}$$

It is important to note that trains do not have the ability to go backwards and therefore need to plan ahead to avoid getting stuck. To learn which actions to take, the agents have to learn to adapt to an unknown environment due to the fact that the environments are randomly generated and differ on each episode. Depending on the given parameters, the size and complexity of the grid can be adjusted.
The goal of each agent is to reach an assigned target train station as fast as possible. Agents that reach this destination are removed from the grid which means, they can no longer obstruct the path of other trains.

Observations

The Flatland environment allows creating observation builders to observe the environment for each agent. While it is possible to observe the whole grid, this does usually not make sense due to the fact that many parts of the rail grid are not relevant to a single train. Flatland offers by default two different observation builders.

GlobalObsForRailEnv creates three arrays with the dimensions of the rectangular rail grid. The first array contains the transition information of the rail grid. For each cell, there are 16 bit values, 4 bit for each possible direction a train is facing.

TreeObsForRailEnv creates a graph with sections of the grid as nodes from the perspective of the train. This means, only the switches which the train is actually able to take define a single node. As an example, a train on a *case 2* switch heading from the top to the bottom is not able to make a decision on this switch and therefore, the TreeObsForRailEnv does not put the sections before and after the switch into two different nodes but just into a single node.
The nodes of the tree observation offer a number of fields that allow selecting specific features to create numeric input vectors for function approximators such as neural networks. The tree observation builder offers 14 distinct features for each rail section. These include:

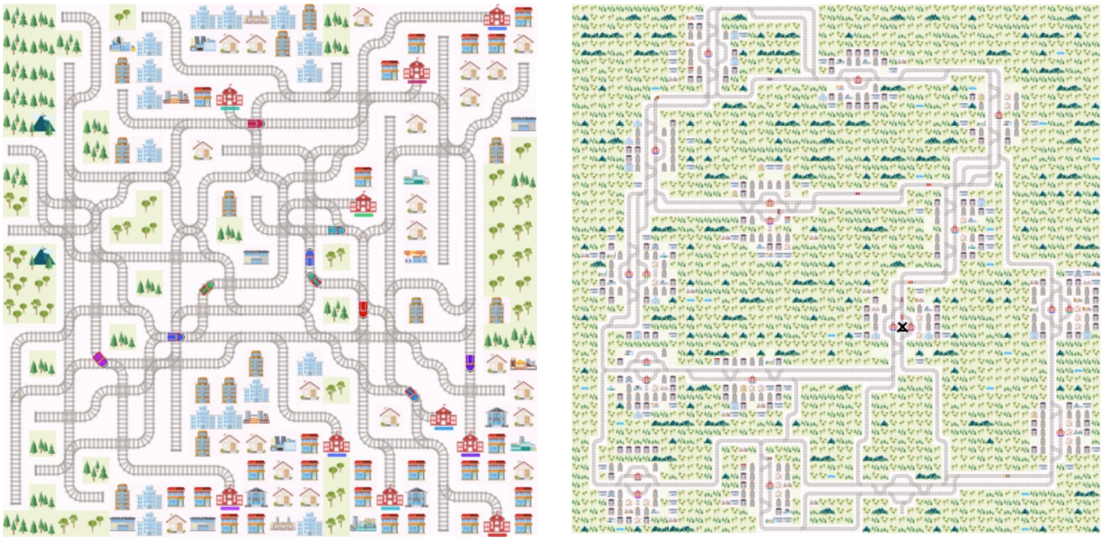
- Distance until own target encountered: Cell distance to the own target railway station. *Infinity* if the target railway station for the agent is not in this section.
- Distance to other agent encountered: Cell distance to the next other agent on this section.
- Distance to next branch: The length of this section in number of tiles.
- Minimum distance to target: The minimal cell distance to the target after this section is finished.
- Child nodes: The nodes the agent is able to take after this section ends. Each child node is associated with a direction (left, forward, right).

Agent Evaluation

AICrowd and SBB also provide a system for agent evaluation. This system evaluates the policy on a number of unknown environments and outputs the percentage of agents that reached their assigned destination as well as the received reward while doing so. The evaluation reward scheme is thereby as follows [11]:

$$\mathcal{R}_t = \begin{cases} -1, & \text{if } s_t \text{ is not terminal} \\ 10, & \text{otherwise} \end{cases}$$

The difficulty of the evaluation level does differ between round 1 and round 2. While round 1 offered many connecting rails between the starting position and the assigned target of an agent, round 2 has more sparse connections and usually only provides 2 to 4 rails between cities. The concept of cities has been added in round 2. Those differences can be clearly seen in Figure 2.4 with the densely connected environment for round 1 versus the sparsely connected environment for round 2.



(a) Evaluation round 1. (b) Evaluation round 2.

Figure 2.4: Comparison between two screenshots from evaluation environments from Flatland round 1 and round 2.

3 Basic Implementation

3.1 A3C Implementation

Originally, the Asynchronous Advantage Actor-Critic Algorithm has been designed for use in a single agent environment [3]. By applying it in a multi-agent environment, we implicitly convert the environment into a non-stationary system. While applying A3C in a multi-agent setting, the other agents can be viewed as part of the environment. This means, the behavior of the environment changes while training, due to the fact that the behavior of the other agents changes. Gupta et al. find in [12], that RL methods like Deep Q-Networks and Trust Region Policy Optimization are not performing well in a multi-agent environment, due to the combination of experience replay and non-stationarity of the environment. Unlike the baseline version from [2], we do not use an experience replay buffer with older episodes. Otherwise the sampled experience might represent old agent behavior which is then learned. Instead, we directly perform updates of the neural network using the latest episodes, as proposed in [3].
An important implementation detail in our version is, that we do not perform updates during episodes but only at their end. The reason for this is, that the only possibility for an agent to receive reward is at the end of the episode. Therefore any previous update would have a reward of 0 and would not help the training process.

3.2 Entropy Balancing

In RL, it is of great importance to find the right combination of exploration and exploitation [13]. During exploration, the agent explores as much of the state space as possible. This enables the agent to later exploit the found states that are beneficial. Without this exploration phase, there is a chance that the agent settles on sub-optimal policies too quickly and ignores parts of the state space the agent has never seen and therefore does not consider in his action selection. To avoid an early convergence in A3C, it is common to use an additional entropy term. This entropy term is defined as [3]:

$$H(p) = -\sum_{u=0}^{U} \log(p_u) \cdot p_u$$

Where p is a possibility distribution over all available actions $\mathcal{U}$. The entropy is multiplied with a factor β and added to the policy loss. This has an effect of preventing a convergence towards a single action, especially in the early phase of training. Without this entropy term, we often observe such an early convergence with grave consequences for the training performance. We start training with $\beta = 0.0025$. This value proved to offer a nice balance between exploration and exploitation in our case. We also observe, that it is recommendable to reduce this factor β to zero once a stage of training is achieved, in which the main objective of the training is not to find the fastest path but to evade collisions with other agents.

3.3 Observation Design

The Flatland environment provides a base to build custom observation builders that can be used to create a state representation for the agents as explained in section 2.2. In this work, we do not consider the usage of the grid-based observation builders. Both the Flatland development team as well as S. Huschauer find, that tree-based observations work better in their experiments [2]. The Flatland specification states [14]:

> Considering, that the actions and directions an agent can choose in any given cell, it becomes clear that a grid-like observation around an agent will not contain much useful information, as most of the observed cells are not reachable nor play a significant role in for the agents decisions.

Based on the provided TreeObsForRailEnv (see section 2.2), we implement a custom observation builder that we use to produce an input vector for our neural network. This observation builder takes the current state of the environment and produces a fixed size numeric vector with values between 0.0 and 1.0 for each agent. This input vector should fulfill a number of requirements:

- Each rail section the agent possibly rides on next should be visible to the agent.
- The agent should be able to detect, whether there is another train coming the opposite direction on any section.
- The agent should be able to detect on each switch which turn is the faster way to his target.
- On switches, the agent should be able to see if a turn does lead to his target, even if it is not the fastest way. If this is the case, taking this turn might even be a good option to evade possibly blocking situations.
- For the next grid tile, the agent should be able to detect if it is a switch and if so, if it is one the agent can make a decision on. (see section 2.2 for non-usable switches).

The provided TreeObsForRailEnv produces a graph with a node for each section of the rail. We extend these nodes with additional information about train traffic coming the other direction than the one the agent is heading. We take the information from these nodes and convert them into a numeric vector. In case that there is a dead end, we fill the observation with zeros (this could only happen in Flatland round 1, round 2 does not have dead ends). After the conversion, we concatenate all section observations into one large vector with information for all upcoming sections as illustrated in Figure 3.1.

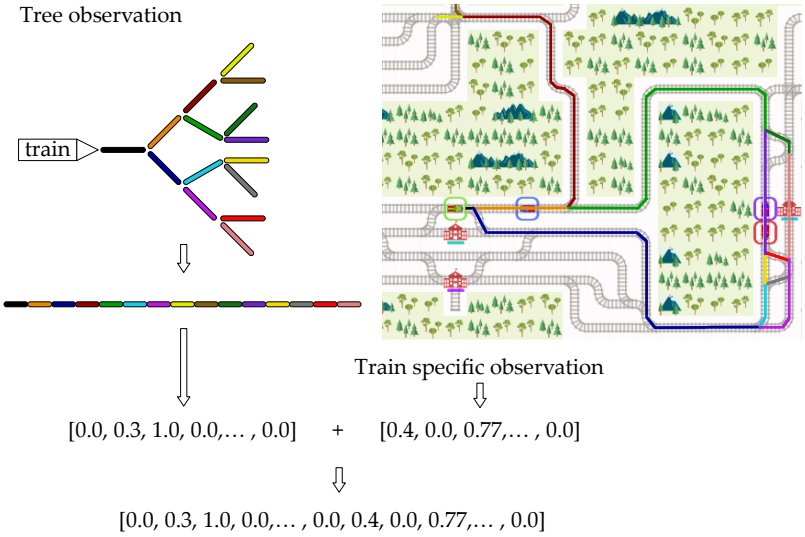


Figure 3.1: Illustration of the tree observation mapped onto the Flatland environment. Each colored bar represents information about a section.

While this vector already contains all required information outside of the agent, we add another vector with information regarding the agent itself (train specific observation in Figure 4.6). This vector contains the speed of the agent, the max. speed of the agent, the type of the current tile, the direction the agent is heading and the type of tile the agent is currently located on. The same idea is used by Bacchiani, Molinari and Patander in [4] with the difference that in their work, the environment observation has no tree structure.

3.4 Technical Implementation Aspects

In this section, we discuss the most important implementation aspects of our work.

Used Frameworks and Libraries

In Table 3.1, the most important libraries and frameworks of this work are listed.

Technology	Version	Description
Cython	0.29.14	Cython allows to compile Python code to natively running C code. We use Cython to speed up our predictions and observations of the rail environment.
Flask	1.1.1	We use Flask as our central model server (see Distributed Architecture and Parallelism). This Flask application collects gradient updates, distributes the current network weights and allows access to the latest versions of the observation and parameter files.
MessagePack	0.6.2	MessagePack allows very fast serialization. The resulting binary data is also considerably smaller than serialization with formats like JSON. We use MessagePack to serialize both weights- and gradient-update before sending between worker and model server.
Python	3.6	The whole code for our project is written in Python.
Tensorflow	2.0.0	Tensorflow provides a framework for both deep learning as well as monitoring the learning process of RL using Tensorboard.
ZLib	(built-in)	ZLib allows to compress data in Python. We use ZLib to reduce our payload-size while sending gradient- and weight updates.

Table 3.1: The most important technologies used in this work.

Usage of Multiprocessing

Due to performance limitations of multithreading in Python, we use the multiprocessing library for parallel execution instead. While multiprocessing is harder to manage than multithreading, the performance limitations of the global interpreter lock [15] do not apply.
The worker processes that are used for training are started by a single master process. This master process handles all required initialization routines as well as restarting the worker processes in case of an error. The usage of parallel processing is described in 4.3 Distributed Architecture and Parallelism.

4 Experiment Design and Analysis

4.1 Reproducibility and Experiment Setup

It is important to note, that the training process of reinforcement learning and especially multi-agent reinforcement learning can be hard to reproduce. Depending on the initial weights of the neural networks and the layout of the environments, the performance may vary on each restart. Also in a distributed algorithm, the number of workers can significantly influence the training performance. If not differently specified, we executed all presented experiments on machines with the same specifications (see section 4.3).
Another aspect that is hard to reproduce is training stability. In A3C, an important instrument to prevent the policy from converging too early is using an additional entropy term [3]. Our way to maintain stability with changing environments is discussed in section 3.2.
For better comparability, we keep our evaluation versions for the training as similar to each other as possible. If not differently specified, we run evaluations with the following setup:

- For all evaluations, a Flatland round 2 environment is used.
- We use an environment with the size 100x100 tiles with 14 individual agents.
- The map contains 20 cities
- There is a maximum of 3 rails between cities and a maximum of 4 rails inside cities
- Based on the Flatland specification, the maximum allowed number of timesteps is 1608 [14].
- For each experiment, we use the training data of the first 12 hours.

To analyze the performance of the solution, we run an analyzer that executes 20 roll-outs of the environment using the same neural network parameters. After these 20 roll-outs, we update the neural network parameters. For each evaluation round, we use the same 20 environment layouts. To compare the performance in a graph, we take the mean number of agents that arrived at their target and plot that in our graph.

4.2 Reinforcement Learning for Flatland

Action Space Reduction and Script Policy Actions

The Flatland environment is designed in a way to resemble a classical RL environment. This means, on every timestep, we receive observations for each agent, calculate an action and hand this action to the environment, visible in pseudo-code in algorithm 1.

Algorithm 1: Default episode for Flatland environment

Data: initialized Flatland environment $\mathcal{E}$, initial observation $s^{a}_{t=0}$ for all agents
Result: terminal Flatland environment
initialize buffer $\mathcal{B}$
while *episode not terminal* **do**
  create empty action array $\mathcal{A}$
  for *every agent a* **do**
    get current state s^{a}_{t} of agent
    // Fetch action for agent, based on current state
    $\mathcal{A}[a] \leftarrow$ from policy $\pi(s^{a}_{t})$
  end
  call *env.step*($\mathcal{A}$)
  retrieve reward $\mathcal{R}$
  append $\mathcal{A}$ to buffer $\mathcal{B}$
  retrieve all new states s_{t+1}
end
use buffer $\mathcal{B}$ for training of policy π

While this makes sense in an environment where agents need to take an action on every timestep (such as Atari games), in Flatland most of the time the only reasonable action is to move forward as visible in Figure 4.1. Only around switches, an agent needs to take actions other than just keep going forward.

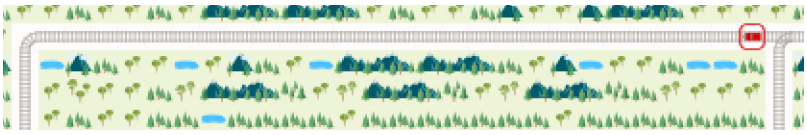


Figure 4.1: Screenshot from Flatland environment. A train heading to the left. The only reasonable action is to ride forward.

Every action that is produced by the neural network should be included for training, so the network can adapt to this type of situation. The problem arises now, that all these actions of riding forward are included in the training of the agent. The influence of the actions that actually matter (e.g. the ones around switches) is thereby not as big as it could be, because large portions of the training data are situations that do not actually require decisions.

Experiment Setup

To solve this problem, we implement hard-coded rules that the agents follow as long as they are not in a situation to make a decision. Only around switches, the neural network policy is activated. As a consequence, the data used for training has fewer samples but the samples available are of higher quality. The training with this mechanism is shown in algorithm 2. For training, we only use the experience collected near the switch.

Algorithm 2: Improved learning algorithm for Flatland environment

Data: initialized Flatland environment *env*, initial observation $s^{a}_{t=0}$ for all agents
Result: terminal Flatland environment
initialize buffer $\mathcal{B}$
while *episode not terminal* **do**
  create empty action array $\mathcal{A}$
  for *every agent a* **do**
    if *agent is near to a switch* **then**
      get current state s^{a}_{t} of agent
      // Fetch action for agent, based on current state
      $\mathcal{A}[a] \leftarrow$ from policy $\pi(s^{a}_{t})$
    else
      $\mathcal{A}[a] \leftarrow u_{forward})$
    end
  end
  call $env.step(\mathcal{A})$
  retrieve reward $\mathcal{R}$
  if *agent was near switch* **then**
    append s_t, $\mathcal{A}$ and $\mathcal{R}$ to buffer $\mathcal{B}$
  end
  retrieve all new states s_{t+1}
end
use buffer $\mathcal{B}$ for training of policy π

Experiment Analysis

This drastically improves training performance as visible in Figure 4.2. While both versions have the potential to perform well, based on the provided observation data, we observe that the version without the action reduction is not able to perform nearly as well as the improved version with reduced actions.

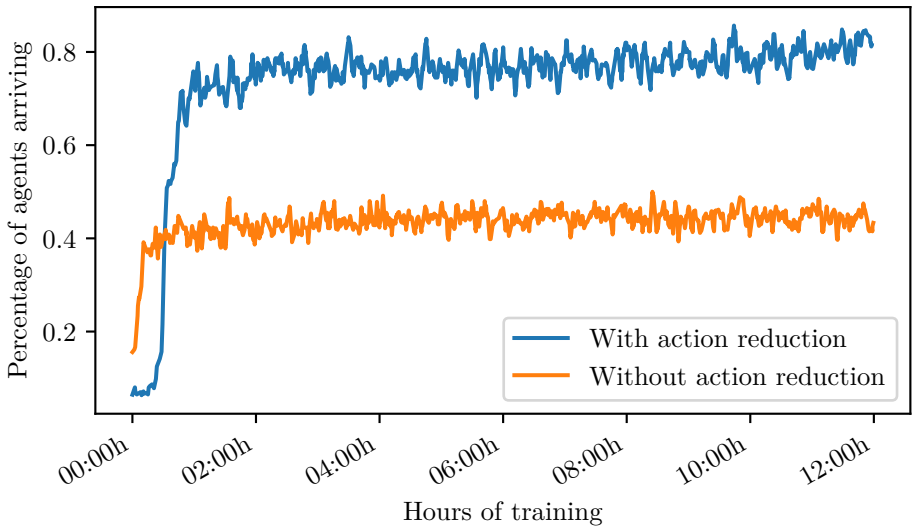


Figure 4.2: Comparison of training with and without action reduction

Neural Network Architecture

In RL, the architecture of neural networks is often rather simple [3, 6]. The already often sample inefficient process of reinforcement learning should not be slowed down by a difficult architecture. We observe a stable training performance by using 3 fully connected layers of size 128, 64 and 64 with ReLU activation for both the actor and the critic network. Differently than S. Huschauer in [2], we do not use convolutional layers. The reason for this is the way our observation vector is composed (see section 3.3). In this vector, every element corresponds with an actual information in the mapped rail section. Convolutional layers would especially make sense if there were patterns to extract from the observation vector.
Another relevant aspect of neural networks is the topic of recurrent layers. Recurrent layers allow the agent to remember information from previous timesteps. The original A3C publication shows, that it is possible to achieve a significant performance improvement using long short-term memory layers (LSTM) [3]. Also S. Huschauer uses an LSTM-block to improve training [2].

Experiment Setup

To quantify this improvement, we run an experiment to compare a version with LSTM layer to a version without. Both versions use the default evaluation environment without curriculum.

Figure 4.3: Comparison of training performance between a version with an LSTM-layer and a version without. It is evident that the version with LSTM performs better than the version without.

Experiment Analysis

In Figure 4.3 it can be observed that an LSTM-layer helps the training process. While the initial training phase is slightly faster for the version without a recurrent layer, we observe a better overall performance and therefore keep the LSTM-layer as part of our solution.

Curriculum Learning and Reward Assignment

The reward assignment in Flatland can be freely configured. But as long as there is not some distance-to-target dependent reward function, the probability that an agent with an uninformed policy finds its target station is small. This is especially the case for large environments with many trains on it. For example, most evaluation environments of Flatland round 2 have up to 200 individual agents and are up 100x100 tiles large [11]. The roll-out of such an environment takes a lot more time than the roll-out of a 20x20 environment with 5 trains. Also, the probability, that a train arrives in a small environment is larger and therefore, the experience is more valuable for training. To improve training times, it makes therefore sense to start with a small environment and move to larger ones, once the agents mastered pathfinding and basic collision avoidance.

Experiment Setup

In order to verify the meaningfulness of such a curriculum, we run an experiment with a large environment with dimensions 100x100 and 50 individual agents. One experiment tries to learn its policy directly using this environment. The other experiment uses a curriculum that gradually gets more difficult. All parameters of the used curriculum are listed in Table 4.1.

	Level 0	**Level 1**	**Level 2**	**Level 3**	**Level 4**
Next level on success rate	70%	70%	75%	70%	60%
Nr. of agent	4	8	12	16	20
Env. size	25x25	30x30	40x40	50x50	50x50
Num. cities	5	8	10	12	16
Max. rails between cities	1	2	2	2	2
Max. rails in city	2	2	3	3	3

Table 4.1: Curriculum level specifications for our experiment to compare a version with curriculum to a version without.

Experiment Analysis

As visible in Figure 4.4, the version without curriculum is not able to notably learn from the experience. We suspect that is caused by too much variance in the environment and not enough successful experience to learn from. After all, an agent needs to reach its target to get reward, and in a large environment combined with an uninformed policy, this is potentially very difficult.
For the version that uses a curriculum, it appears that the learning speed is slowed down on higher curriculum levels. With increasingly more difficult levels and more agents, the frequency of available network updates is being slowed down due to the fact that a roll-out of the environment takes longer. We therefore think, it is desirable to keep the curriculum levels small and primarily increase the number of agents instead of the size of the environment.

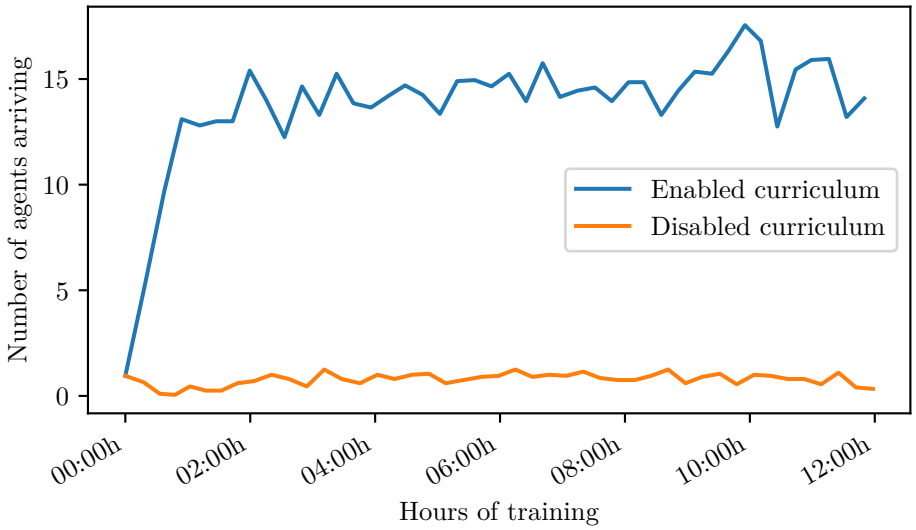


Figure 4.4: Comparison of training performance between a version with and a version without curriculum learning.

Agent Communication

Communication in multi-agent reinforcement learning is a topic of active research [16]. While we do not use communication in the final submission, in this section we discuss a number of ideas regarding possible the use of communication in the Flatland environment. As a starting point for this work, we asked the question how blocking is avoided in the real-world. On the SBB rail environment railroad signals are used to prevent collisions between trains. These signals are controlled by a central control instance that directs the flow of traffic for all trains on the network. The Flatland environment does not provide a communication channel nor a rail road signal system, but participants of the challenge are allowed to implement one themselves. In the following discussion, we summarize available options to implement such a communication channel. An initial intuition for the problem gives the question, if it was a human guiding the train, would she/he be able to successfully steer the train to its assigned target without communication? We assume that this is not the case. While classical traffic rules (e.g. right has priority over left) can help to avoid collisions, we do think that in the case of railroad traffic this does not suffice. In the real-world, rail traffic is controlled by a control center that gives the trains permission to go or tells them to wait. Having one central control instance solves one of the big problems of such a system which is, that there is a chance that two agents take an action at the same moment which brings both of them into a non-resolvable situation. An example for such a situation would be two agents that enter a rail section at the same timestep. Both of them would have had the possibility to choose a different path, but both perceived the section as empty and decided to enter. Combined with the fact that trains cannot drive backwards (at least in Flatland), both trains will not be able to arrive at their assigned target stations. As a solution to this dilemma, we discuss four ideas that could potentially improve the situation.

- **Negotiation:** To solve the problem of conflicting actions at the same timestep, it would be possible to introduce an iterative communication channel, on which the agents can negotiate, what agent is allowed to take an action next and who has to wait for another timestep. The number of iterations could be determined by the outcome of the process. As long as there is no resolution about who is allowed to take an action and who is not, another negotiation round is added.
  While this procedure might help to avoid blocking situations, it could certainly not completely remove them. Especially complex situations with many agents in a small area could still prove to be difficult, even with a negotiation mechanism in place.
- **Prioritized planning:** An approach to solve the dilemma of conflicting actions at the same timestep could be to introduce an artificial order of importance among the agents. Then the next n timesteps could be planned for the most important agent. The second agent now takes the planning of the first agent into account and tries to come up with a plan that does not obstruct the planned route of the first agent. This process is repeated for all following agents.
  We think that possible conflicts could probably be resolved by backtracking and adjusting the priorities of the trains.
- **Unconstrained communication:** While negotiation would prevent an agent from taking an action, the goal of unconstrained actions would not be to constrain the action space but rather to convince the agent to choose an action *stop* or *do nothing* and wait for the next timestep if the situation is uncertain. Also for unconstrained actions, it would make sense to be an iterative process, similar to the negotiation approach. As long as not all agents mark their communication as completed, another communication round is added.
  We suspect that a problem with this approach could be that it would have much slower convergence than the negotiation approach. The reason for this assumption is the fact, that such an unconstrained communication would not directly influence the actions of the agents. It would be necessary to learn both the ”speaking” and the ”interpretation”, without direct consequences. On the positive side, we think that this approach could enable more sophisticated strategies and maybe even improve planning.
- **Announcing communication:** Different from unconstrained communication, under *announcing* communication we mean agents that announce their next action over a shared communication channel. As a metaphor, one can imagine a bus on a narrow mountain street honking before each turn. A mechanism like this would, similar to *prioritized planning*, require to receive the agent actions in a sequential way, so that each following agent could react to the actions of the previous agents.

We think that the two areas of prioritized planning and unconstrained communication would have the biggest potential to improve train behavior in our Flatland solution. In this work, we do not consider implementing planning and therefore focus on the case of unconstrained communication. We argue, that a communication channel that does not provide a defined protocol could eventually also have the benefit of allowing the agents to plan ahead.

Experiment Setup

To verify the assumption that it is possible to learn a communication protocol to plan ahead, we create an experiment that reduces the observation space to a shared communication buffer. For this experiment we define a Flatland layout with two train stations and only one rail that connects them. We let a train start in each train station and assign the opposite station as its target. We also add two detour tracks that allow the agents to evade a collision with the agent as can be seen in Figure 4.5.

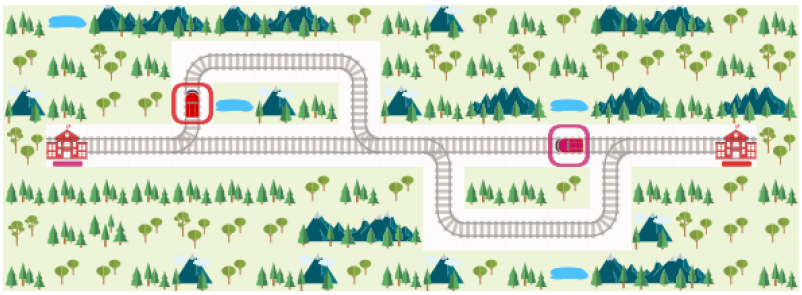


Figure 4.5: Screenshot of the communication experiment setup.

The agents only need to take an action on switches they encounter, otherwise, they just drive forward. The agents can not observe the upcoming sections and only have access to the said buffer. Due to the symmetry of the environment, both agents need to take an action at the same time. The agents can select an action between 0 and 5. On taking an action, we start a communication loop that allows the two agents to alternately read from the buffer, calculate an action and write that action back into the buffer. This loop continues until both agents output the action 5 (= we are done with the communication). After finishing communication, both agents can select an action to take next in the environment. If both agents take the same action and collide, we cancel the episode and give a reward of -1. If they make it around each other and reach their targets, they receive a reward of +1. The agents have no way to know, if they are agent 1 or 2 (otherwise, agent 1 could just always make a loop and agent 2 could go straight).

Experiment Analysis

We observe, that the agents are able to learn a protocol themselves. After 100'000 episodes of training, the success rate to fulfill the task is as high as 95%. Previously to the experiment, we assumed, that if the agents would learn a way of communication, it would probably look the same for every episode. This assumption has not been confirmed. In all observed cases, the agents need between 2 and 5 communication timesteps. In Table 4.2, we list 6 examples of observed communications with their regarding outcomes.

Timestep	**Actions agent 1|2**	**Outcome**
0	4 | 2	
1	5 | 5	Success
0	3 | 0	
1	1 | 5	
2	5 | 5	Success
0	3 | 5	
1	5 | 5	Success
0	3 | 1	
1	3 | 2	
2	5 | 0	
3	5 | 5	Crash
0	3 | 2	
1	5 | 3	
2	5 | 4	
3	2 | 5	
4	5 | 5	Success
0	4 | 3	
1	3 | 1	
2	5 | 5	Success

Table 4.2: Examples of communication in our agent communication experiment.

While we are surprised that the communication does not look the same on every episode, we think it is remarkable how apparently a language between the agents emerges. Due to reward discounting, it would make sense to have as few communication steps as possible. Still, the agents seem to decide, that it is more valuable to add more communication steps to reach an agreement, instead of going straight to the end of communication.
It is also interesting that agent 2 never seems to respond with the same action as agent 1. The question, if such a behavior could be learned in a less constrained environment is discussed in 6 Discussion and Outlook.

4.3 Distributed Architecture and Parallelism

Distributed Training

One of the main advantages of the A3C algorithm is its ability to be used in a distributed manner. This allows running multiple versions of the environment asynchronously and collect the updates for both the actor and the critic network in a central place. To fully take advantage of this mechanism, we implemented a system that allows to run a large number of environments at the same time. All running training-instances contribute to the same central network. Thanks to HTTP-based connections, we can use this mechanism even between computers or across networks. This approach is only limited by the capability of the central model server and the network throughput.
Additionally, our central model server not only handles the update of the neural network but also distributes the code for building observations, predictions and a file with all hyperparameter required for training on startup. The startup process of the worker node then converts the code for building observations and predictions into native C-code using Cython. This converted code then gets compiled and dynamically imported as a Python module.
Using native C-code speeds up the roll-out of the environment and therefore the training process by a factor of 2 to 5.

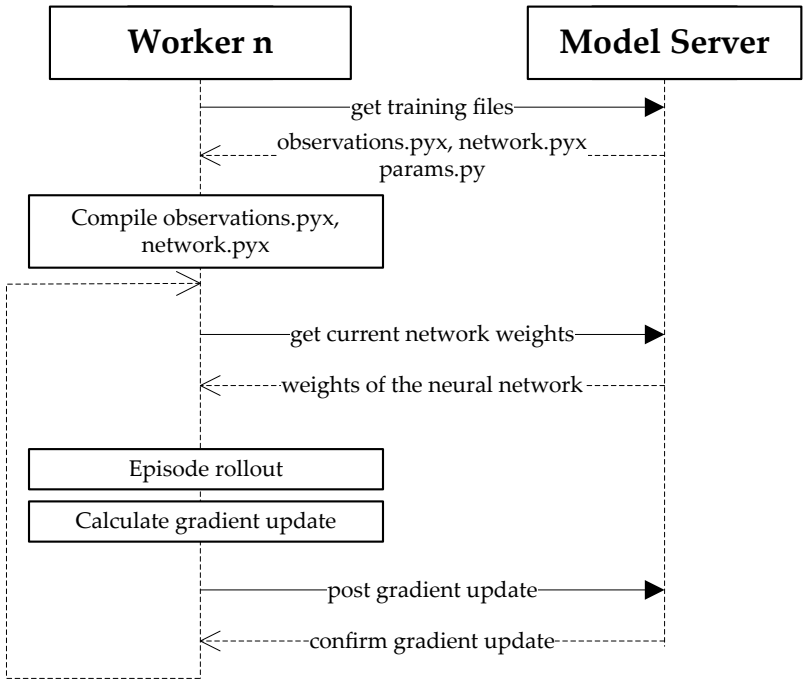


Figure 4.6: Training of a single worker process. The communication between the worker and the model server works by using HTTP. The compilation of the .pyx-files is only done once on each machine.

To improve the performance of the system, we compress both the weights (server to the worker) and the gradient update (worker to the server) using ZLib. This reduces the size of the transmitted data between 10% to 80%.

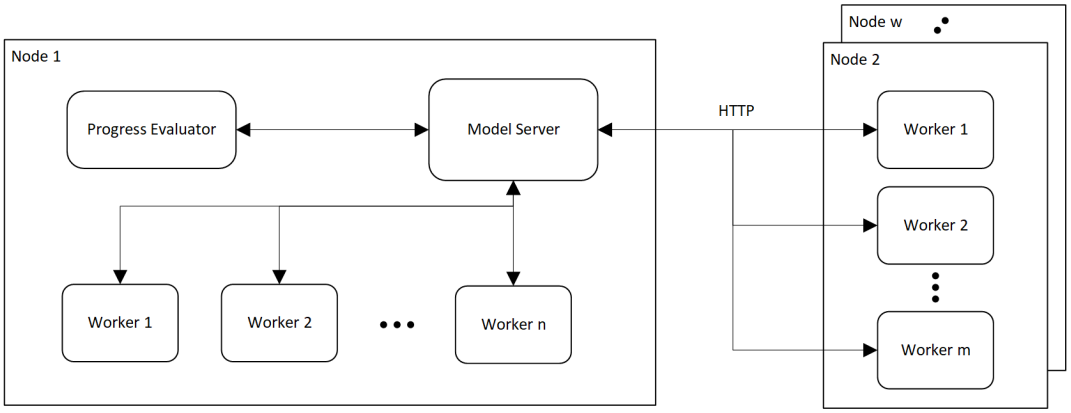


Figure 4.7: Illustration of distributed architectures with the main node (node 1) and several additional nodes (2-w) connected over HTTP.

Experiment Setup

To analyze the impact of training a policy with multiple workers at the same time, we run an experiment that compares training with a single worker to training with 7 workers at the same time. We use the default analysis environment that includes reduced actions but no curriculum learning.

Experiment Analysis

While it is apparent in Figure 4.8 that having multiple workers does speed up the training process, we do not observe any impact of the number of workers on the quality of the policy. These results are consistent with the observation that training with even more than 7 workers speeds up training, but has no noticeable effect on the quality of the policy.

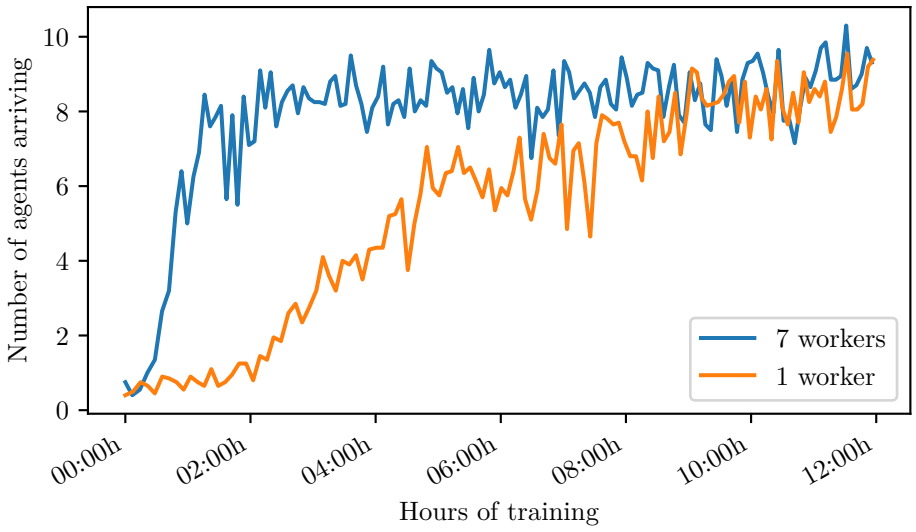


Figure 4.8: Comparison of training with 1 worker to training with 7 workers.

Infrastructure

The infrastructure used for this project consists of 4 machines used in various combinations.

- **Draft Animal (DRAN):** Server with 56 CPUs and 721 GB of RAM. This machine is mainly used to train the current submission model.
- **Openstack machines:** 3 servers with 8 CPUs each. One machine has 16 GB of RAM, the other two have 64 GB each.

The A3C algorithm is not able to take advantage of GPUs. The reason for this is the fragmented update process. Differently than with an experience replay buffer, the training data is only used once and gets then discarded.

5 Results

5.1 Round 1

Our submission for Flatland round 1 does not include all algorithmic improvements discussed in this work. None the less, we were able to achieve a significant improvement in performance compared to the baseline version from [2]. Our submission for round 1 contains the following components:

- Custom A3C implementation without experience replay buffer.
- Default TreeObsForRailEnv (in round 1, this was a numeric vector by default).
- Policy learned by curriculum learning.
- Distributed training over multiple processes on the same machine (no cross-machine distribution possible yet).

Using the performance evaluation system provided together with the Flatland environment, we reach the following performance metrics:

Author	Observation type	Local Evaluation Score	Submission Score
S. Huschauer	reduced grid observation	19.4%	16.6%
	tree observation	24.7%	-
Meier/Roost	tree observation	69.3%	48.9%

Table 5.1: Comparison of submission performance for Flatland challenge round 1.

5.2 Round 2

Our submission for Flatland round 2 contains all in this work discussed improvements of the algorithm except communication. While a direct comparison is difficult due to a lack of a baseline version for round 2, we could still show in our experiments how our adjustments to the implementation improved the performance of our solution.
To evaluate the performance of the final model, we create an experiment with an increasing number of agents. We use an environment of the dimensions 30x30 tiles and keep all other environment parameters to a fixed value. Only the number of agents gets increased. For each number of agents, we run 20 episodes and evaluate the median, the 0.25-quantile and the 0.75-quantile.

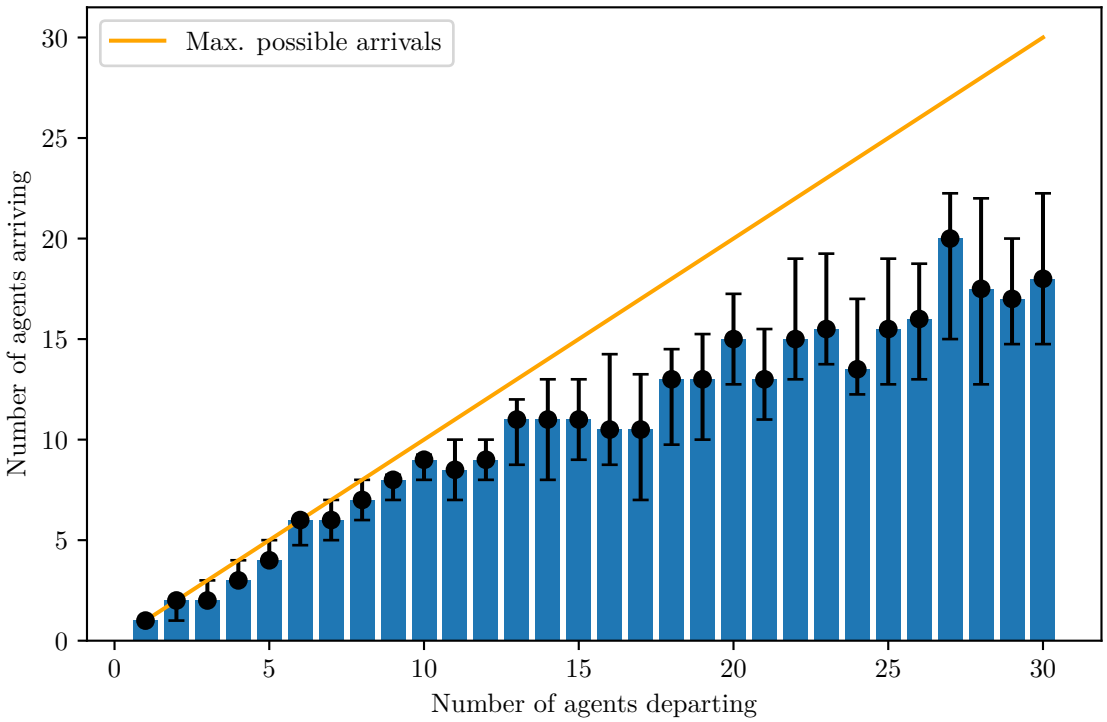


Figure 5.1: Agents departing vs. agents arriving on a 30x30 environment.

In Figure 5.1, it can be observed that our model is performing very well for environments with 10 agents or less. For environments with more agents, the performance gradually gets worse. It is important to note that this change in performance is not only dependent on the absolute number of agents but also on the density of agents relative to the size of the environment.
The evaluation system for Flatland round 2 uses 250 different environments with sizes between 20x20 to 150x150 [11]. The number of agents varies from 50 to 200. Evaluated with the Flatland round 2 evaluator, we achieve a score of **29.1%** of arriving agents. At the time of submission, our submission performs the 4th best out of 24 contestants that have submitted a solution for round 2 at this point. Please note, that this is not the final ranking of the Flatland challenge.
When analyzing our solution in selected example environments, it can be observed that agents especially struggle in two cases:

1. Two agents have to take a decision at the same time.
2. Agents deadlock each other in areas with many switches.

Possible solutions for the listed problems are discussed in both section 4.2 and chapter 6.

6 Discussion and Outlook

6.1 Review of the Application of Reinforcement Learning

While trying to solve the Flatland challenge with RL, we discovered a limitation of policy gradient-based methods in high-consequence environments like Flatland. We call Flatland a high-consequence environment because taking a bad action quickly leads to a chain reaction of unresolvable situations. Policy gradient-based RL uses a probability distribution over all available actions. The difficulty of combining a high-consequence environment with a stochastic policy can be nicely shown in an example: We consider a situation where the best action is taken with a probability of 90%, the remaining (probably non-beneficial actions) have a combined chance of 10% to be selected. For 10 agents with such a probability distribution, there is already a 65.1% chance that one of the agents takes a non-beneficial action and creates a chain reaction of problems. Just converting this probability distribution into a deterministic policy by using the *argmax* over the probability distribution does not solve the problem due to some situations in which the agent is not sure what to do and therefore assigns similar probabilities to the actions. The algorithm then relies on its stochasticity to try all available actions. Therefore it would be interesting to experiment with value-based RL algorithms to observe, if such algorithms can overcome the described problem that policy gradient methods have. In most popular use-cases of RL such as Atari games, it will suffice to just select a good action and not necessarily the best. This is different in Flatland and should therefore be addressed.

6.2 Practicability in a Real-World Scenario

While we were able to greatly improve the performance of the presented solution compared to the given baseline, it is still nowhere near practical applicability. While the presented evaluation tasks probably do not represent the real-world density on a rail network, also with a lower volume of traffic, train traffic would require more robust solutions with the primary objective of finding a solution for every train to reach its destination instead of optimizing the performance of a single agent.

6.3 Ideas for Future Research

We think that in order to further improve performance, the problem would need to be formulated in a different way. While the research presented in this work is mainly focused on treating the trains as agents in a multi-agent reinforcement learning problem, it might be an interesting approach to introduce a central planning agent that takes over all planning in advance. Especially in a simulated environment like Flatland with perfect information, we think that upfront planning would have the potential to take full advantage

of the available data and the possibility to iteratively plan the upcoming steps. With all planning done by a single agent, it would also remove the requirement for communication. While we could show in section 4.2 that it is possible to learn communication protocols between agents, we think that in a less constructed example the convergence towards a usable communication protocol might be too slow to actually use it in real-world training with many agents.
Combined with the possibility of a centrally planning agent, we think that converting the rail network into a logical graph would enable any training algorithm to perform better without the need for respecting the tile-based architecture of the Flatland environment. In section 4.2, we already take a step in this direction by reducing the actions required for each agent.
An alternative to the centralized planning approach could be something similar to the solution presented by Ephrati and Rosenschein in [17]. They propose to plan on a local level and merge the locally found solutions into a global plan. Using the presented subgoals it could be possible to resolve local conflicts and iteratively move into a direction of a global solution.

7 Listings

7.1 Bibliography

[1] E. Nygren, S. Mohanty. Flatland challenge. [Accessed: 2019-11-13]. [Online]. Available: https://www.aicrowd.com/challenges/flatland-challenge

[2] S. Huschauer, "Multi-agent based traffic routing for railway networks."

[3] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[4] G. Bacchiani, D. Molinari, and M. Patander, "Microscopic traffic simulation by cooperative multi-agent deep reinforcement learning," 2019.

[5] E. Nygren, S. Mohanty, C. Baumberger, C. Eichenberger, A. Egli, M. Ljungström, G. Mollard, G. Spigler, J. Watson. Flatland documentation. [Accessed: 2019-12-01]. [Online]. Available: http://flatland-rl-docs.s3-website.eu-central-1.amazonaws.com/

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013. [Online]. Available: https://arxiv.org/pdf/1312.5602.pdf

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. [Online]. Available: https://science.sciencemag.org/content/362/6419/1140

[8] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," 2019.

[9] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, Aug 1988. [Online]. Available: https://doi.org/10.1007/BF00115009

[10] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, ser. NIPS'99. Cambridge, MA, USA: MIT Press, 1999, pp. 1057–1063. [Online]. Available: http://dl.acm.org/citation.cfm?id=3009657.3009806

[11] E. Nygren. Flatland faq. [Accessed: 2019-12-09]. [Online]. Available: https://gitlab.aicrowd.com/flatland/flatland/blob/master/FAQ_Challenge.md/

[12] J. K. Gupta, M. Egorov, and M. Kochenderfer, "Cooperative multi-agent control using deep reinforcement learning," in *Autonomous Agents and Multiagent Systems*, G. Sukthankar and J. A. Rodriguez-Aguilar, Eds. Cham: Springer International Publishing, 2017, pp. 66–83.

[13] L. Rusch, "Exploration-exploitation trade-off in deep reinforcement learning."

[14] E. Nygren, S. Mohanty, C. Baumberger, C. Eichenberger, A. Egli, M. Ljungström, G. Mollard, G. Spigler, J. Watson. Flatland specification. [Accessed: 2019-12-07]. [Online]. Available: http://flatland-rl-docs.s3-website.eu-central-1.amazonaws.com/04_specifications.html/

[15] Python - global interpreter lock. [Accessed: 2019-12-19]. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock

[16] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "A survey and critique of multiagent deep reinforcement learning," 2018.

[17] E. Ephrati and J. S. Rosenschein, "Multi-agent planning as the process of merging distributed sub-plans," 1993.

[18] Aicrowd: Crowdsourcing ai to solve real-world problems. [Accessed: 2019-12-17]. [Online]. Available: https://www.aicrowd.com/

[19] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, July 1948. [Online]. Available: https://ieeexplore.ieee.org/document/6773024

7.2 List of Figures

7.3 List of Tables

7.4 Glossary

Hint: For abbreviations, see 7.5 Abbreviations.

Term	Explanation
Agent	An agent is the entity that needs to take a decision, based on its current situation. In this work, an agent usually corresponds with a train.
AICrowd	A crowdsourcing platform for competitions about machine learning [18]. The Flatland challenge has been partly organized by AICrowd.
Asynchronous Advantage Actor-Critic Algorithm	A state of the art algorithm for reinforcement learning [3].
Deep Q-Network	A reinforcment learning technique propose by Mnih et. al that uses recorded experience for training [6].
Episode	The cycle of taking actions and receiving a new state until the environment has reached a terminal state.
Entropy	A term for how much randomness can be expected from a variable [19]. For a single variable an entropy of 1 corresponds to complete randomness, and entropy of 0 corresponds to complete certainty. Abbreviated as $\mathcal{H}$.
Environment	A system for reinforcement learning agents to learn. In this work, an environment is usually a Flatland rail simulation.
Hypertext Transfer Protocol	A stateless protocol to transfer data between different applications on the application layer.
Observation	The information extracted from the environment needed to make a decision. Abbreviated as s.
Policy	The rules used to decide which action to take, based on a given state. Abbreviated as π.
Rectified Linear Unit	An activation function for neural networks. It is a piecewise linear function, if the result is positiv the output will be the input otherwise it will return zero.
Reinforcement Learning	See: 2.1 Reinforcement Learning.
Reward	The scalar value an agent can receive in a reinforcement learning task. Positive for good actions, negative for bad ones.
Worker	A process that repeatedly rolls out Flatland environments to create training experience for the reinforcement learning algorithm.
ZLib	A Python module to compress and decompress data.

Table 7.1: Glossary definitions

7.5 Abbreviations

The explanations can be found in 7.4 Glossary.

Abbr	Abbreviation
A3C	Asynchronous Advantage Actor-Critic Algorithm
CPU	Central processing unit
DQN	Deep Q-Network
GPU	Graphics processing unit
HTTP	Hypertext Transfer Protocol
LSTM	Long short-term memory
RAM	Random-access memory
ReLU	Rectified Linear Unit
RL	Reinforcement learning
RSP	Re-scheduling problem
SBB	Swiss Federal Railways
TMS	Traffic management systems
TRPO	Trust region policy optimization

Table 7.2: Abbreviations

8 Appendix

If you are interested in the source code of this work, you can reach us at:
dano.roost@gmail.com or ralphlmeier@gmail.com

8.1 USB Flash Drive Content

The attached USB flash drive contains the following content:

- This report as PDF
- The source code of all experiments
- The source code of the final training (final state)
- The best submission for round 1 and round 2

8.2 Official assignment

Projektarbeit 2019 - HS: PA19_wele_01

Allgemeines:

Titel:	Reinforcement Learning mit einem Multi-Agenten System für die Planung von Zügen
Anzahl Studierende:	2
Durchführung in Englisch möglich:	Ja, die Arbeit kann vollständig in Englisch durchgeführt werden und ist auch für Incomings geeignet.

Betreuer:

HauptbetreuerIn: Andreas Weiler, wele

NebenbetreuerIn: Thilo Stadelmann, stdm

Zugeteilte Studenten:

Diese Arbeit ist zugeteilt an:
- Ralph Meier, meierr18 (IT)
- Dano Roost, roostda1 (IT)

Fachgebiet:

DA	Datenanalyse
DB	Datenbanken
SOW	Software

Studiengänge:

IT	Informatik

Zuordnung der Arbeit :

InIT	Institut für angewandte Informationstechnologie

Infrastruktur:

benötigt keinen zugeteilten Arbeitsplatz an der ZHAW

Interne Partner :

Es wurde kein interner Partner definiert!

Industriepartner:

Es wurden keine Industriepartner definiert!

Beschreibung:

Reinforcement Learning ist der Zweig des maschinellen Lernens, der sich damit beschäftigt, in einer gegebenen Umgebung durch Interaktion automatisch herauszufinden, was das beste "Rezept" (die sog. "Policy") ist, um ein bestimmtes Ziel zu erfüllen. In jüngster Zeit erregten grosse Erfolge der Methodik im automatischen Gameplay (Dota2, QuakeIII, Atari, Go, ...) einiges an Aufsehen. Aber wie die monatlichen Treffen des "Reinforcement Learning Meetups Zürich" zeigen (https://www.meetup.com/de-DE/Reinforcement-Learning-Zurich/), gibt es auch immer mehr vielversprechende Anwendungen in Industrie und Wirtschaft.

Die Hauptfrage bei dieser Arbeit ist: Wie können Züge lernen, sich automatisch untereinander zu koordinieren, um die Verspätung der Züge in grossen Zugnetzwerken zu minimieren. Die Betreuer dieser Arbeit haben bereits eine enge Zusammenarbeit mit der SBB zu diesem Thema aufgegleist, die als Grundlage den gerade gemeinsam ausgeschriebenen KI Wettbewerb "Flatland Challenge" hat (siehe Link unten). In dieser Projektarbeit geht es darum, einen (Deep) Reinforcement Learning Ansatz für Flatland zu implementieren und zu evaluieren.

Informations-Link:

Unter folgendem Link finden sie weitere Informationen zum Thema:
https://www.aicrowd.com/challenges/flatland-challenge

Voraussetzungen:

- Spass an der Arbeit mit Daten und Data Science Tools
- Starkes Interesse am Thema Künstliche Intelligenz, insbesondere Reinforcement Learning
- Sehr gute Programmierfähigkeiten (Python-Kenntnisse können im Projekt erworben werden)
- Pragmatisches und systematisches Vorgehen beim Experimentieren und genauen Auswerten
- Freude am wissenschaftlichen Arbeiten und den ersten eigenen Versuchen in angewandter Forschung

Die Betreuer haben viel Freude am Thema und mehrere Ideen zum Starten auf Lager; sie freuen sich auf leistungsfähige Studierende und ggf. (bei guten Resultaten) eine gemeinsame wissenschaftliche Publikation aus der Zusammenarbeit.