Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Project thesis (Computer Science)

Reinforcement Learning mit einem Multi-Agenten System für die Planung von Zügen

Authors	Dano Roost Ralph Meier
Main supervisor	Andreas Weiler
Sub supervisor	Thilo Stadelmann
Date	18.09.2019

Zusammenfassung

Zusammenfassung in Deutsch

Diese Arbeit befasst sich mit der Steuerung von komplexen Zugverkehrssystemen mittels Reinforcement Learning.
Die Aufgabenstellung hat die Schweizerische Bundesbahn (SBB) im Rahmen einer Challenge auf AICrowd ausgeschrieben. Im Fokus steht eine Lösung zu entwicklen, welche die Verspätung bei technischen Störrungen oder defekten Zügen verringert und es ihnen zudem ermöglicht in Zukunft mehr Züge auf die gleiche Infrastruktur zu bringen. Aufgrund der steigenden Anzahl an Pendlern in den letzen Jahren hat die Erhöhung der Menge der Züge auf dem heutigen Schienennetz eine hohe Relevanz. Die SBB stellt in Zusammenarbeit mit AICrowd die Simulationsumgebung Flatland zur Verfügung. Mittels Flatland kann ein komplexes Schienennetz simuliert werden, mit welchem die Agents (Züge) interagieren können. Es handelt sich hierbei um ein kollaboratives Muli-Agent Problem, welches mit Machine Learning genauer gesagt Reinforcement Learning gelöst werden sollte. Als Basis dieser Arbeit diente uns die Vertiefungsarbeit 2 von S. Huschauer, welche zu der gleichen Challenge gemacht wurde. Wir konnten die Auswahl des A3C Algorithmus und einige weitere Überlegungen von ihm übernehmen. Wir bauten die Lösung von S. Huschauer nach und erweiterten diesen Stand nach und nach mit unterschiedlichen Features wie Long short-term memory, Curriculum Learning oder Parallelisierung. Wir kontrollierten unsere Änderungen mittels Experimenten, welche die Auswirkungen aufzeigten. Durch das Hinzufügen von unterschiedlichen Features konnten wir ein Resultat von 48.9% in Runde 1 erzielen. In Runde 2 veränderten wir den Aktionspace, was die Anzahl Abfragen vom neuronalen Netzwerk deutlich reduzierte. Dies verbesserte die Laufzeit sowie den Lernfortschritt stark. Durch den Einsatz von Regeln, welche spezielle Fälle verhindern, konnten wir eine Ankunftwahrscheinlichkeit von 29.1% erreichen, was aktuell Platz 4 entspricht.

Abstract

Abstract in English

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 into the right direction.
  It was very interesting to work in the Reinforcement learning field and improve the control of train traffic with machine learning.
- Remo Maurer for providing the infrastructure, especially the infrastructure test server and extending the period uf use until the end of the semester.

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.

Contents

1. Introduction **6**
1.1. Baseline . . . 6
1.2. Goal of this work . . . 7

2. Technical Foundation **8**
2.1. Reinforcement Learning . . . 8
2.2. The Flatland Rail Environment . . . 10

3. Approach and Methodology **14**
3.1. Iterative Work Approach . . . 14
3.2. Reproducibility . . . 14

4. Experiments **16**
4.1. RL for Flatland . . . 16
4.2. Distributed Training . . . 23
4.3. Technical Implementation Aspects . . . 26

5. Results **27**
5.1. Round 1 . . . 27
5.2. Round 2 . . . 28

6. Diskussion and Outlook **30**

7. Verzeichnisse **31**
Literaturverzeichnis . . . 31
(Abbildungsverzeichnis) . . . 33
(Tabellenverzeichnis) . . . 34
(Abkürzungsverzeichnis) . . . 35
(Listingverzeichnis) . . . I

A. Anhang **II**
A.1. Projektmanagement . . . II
A.2. Weiteres . . . II

1. Introduction

1.1. Baseline

- Nennt bestehende Arbeiten/Literatur zum Thema -> Literaturrecherche
- Stand der Technik: Bisherige Lösungen des Problems und deren Grenzen
- (Nennt kurz den Industriepartner und/oder weitere Kooperationspartner und dessen/deren Interesse am Thema Fragestellung)

This work explores a real world usage of multi agent reinforcment learning (MARL) for controlling train traffic in a complex railway system. As part of the Flatland challenge, a contest created by the Swiss Federal Railways (SBB AG) 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 sucessfully reach their destinations. The endeavour 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 transportation 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, less switchover facilities and in general more scheduled trains in a shorter time.

This work is based on the work of Stefan Huschauer [2] and further investigates on 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 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

- Formuliert das Ziel der Arbeit
- Verweist auf die offizielle Aufgabenstellung des/der Dozierenden im Anhang
- (Pflichtenheft, Spezifikation)
- (Spezifiziert die Anforderungen an das Resultat der Arbeit)
- (Übersicht über die Arbeit: stellt die folgenden Teile der Arbeit kurz vor)
- (Angaben zum Zielpublikum: nennt das für die Arbeit vorausgesetzte Wissen)
- (Terminologie: Definiert die in der Arbeit verwendeten Begriffe)

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 it could be argued that there are better ways to solve the given problem than RL, we mainly focus on pure RL but give in the chapter 5 our intuition 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 into 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, the details of the Flatland environment can be found in section 2.2. For a deeper understanding of the complex Flatland environment, it is recommended to study the Flatland documentation and specification [5] as well as the official Flatland introduction [1].

2. Technical Foundation

2.1. Reinforcement Learning

Basic Definitions

In recent years, major progress has been achieved in the field of reinforcement learning (RL) [6],[7],[8]. In RL, an agent learns to perform a task by interacting with an environment $\mathcal{E}$. 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 positive reward from the environment, if it does something bad, there is no or negative reward.

The goal of the agent is now to take an action that maximizes the expected reward for all future timesteps. $\mathbb{E}[\mathcal{R}_{t+1} + \mathcal{R}_{t+1} + \mathcal{R}_{t+1} + ...|s_t]$ given the current observation s.
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.

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.

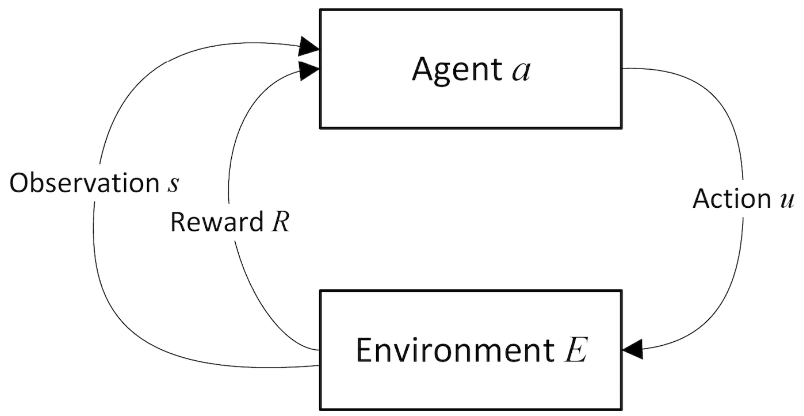


Figure 2.1.: Reinforcement learning overview

Value-based RL algorithms work by learning a value function $\mathcal{V}(s)$ through repeated roll-outs 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 form of a neural network as 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:

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

In some value-based algorithms such as 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.
The second category of reinfocement learning algrithms are the so called policy gradient based methods. These methods aim to aquire a stochastic policy π 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* (A3C), developed by Mnih et al. [3] fits into this category. It uses both a policy π and a value function $\mathcal{V}(s)$. Both are usually seperate function approximators (neural networks in our case).

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

To enhance the process of learning policy π, 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.

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

This means that 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 adjust into a direction that favors the experienced actions based on the seen states.
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.

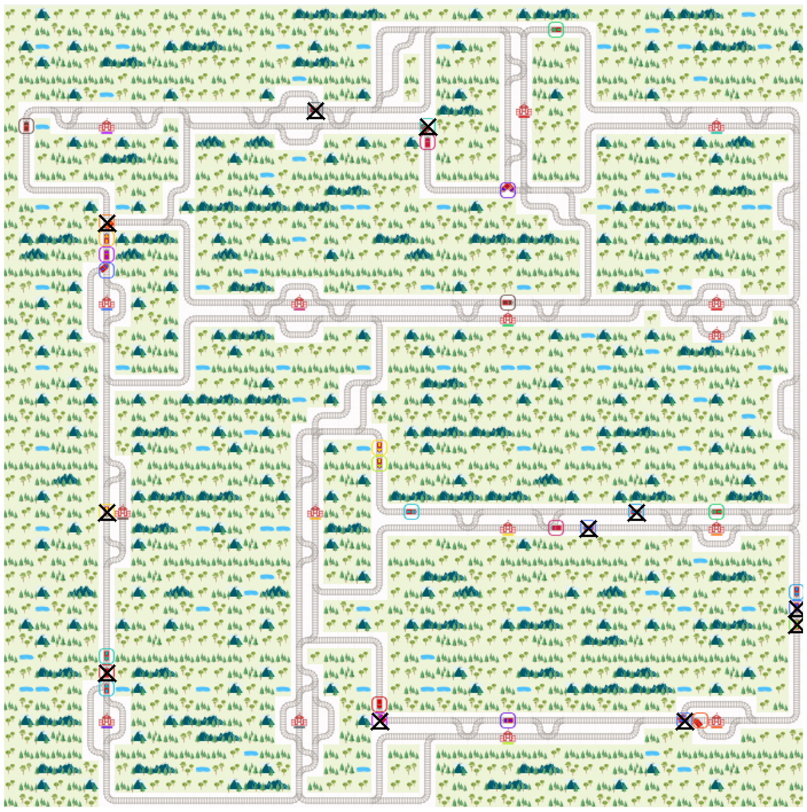


Figure 2.2.: Screenshot from a running Flatland environment.

Relation to this Work

The goal of this work is to apply an RL algorithm to the vehicle rescheduling problem. Based on the work of S. Hubacher [2], we use a distributed RL algorithm that learns a policy to control the traffic of trains on a rail grid. To do so, we use the asynchronous advantage actor critic algorithm [3] and expand its definiton to the use case of multiple agents, similar to [4].

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. Using Flatland, we can train RL algrithms 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 out of squares that can have connections to neighbouring squares. In certain squares, the rails splits into two rails. On those switches, the agent has to make a decision which action it wants to take. Dependent on the type of switch, there are different actions available. 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 north side. All rail parts, independent of if it is a switch also allow to take the actions to do nothing (remain halted, or keep

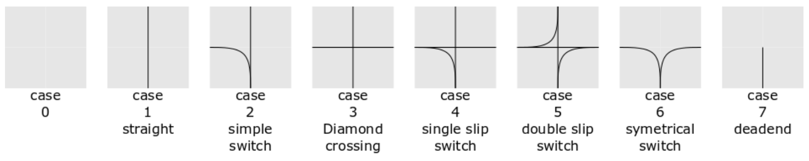


Figure 2.3.: Possible switches in the Flatland environment from [5].

riding), to go forward or to brake. The action space is therefore defined by:

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

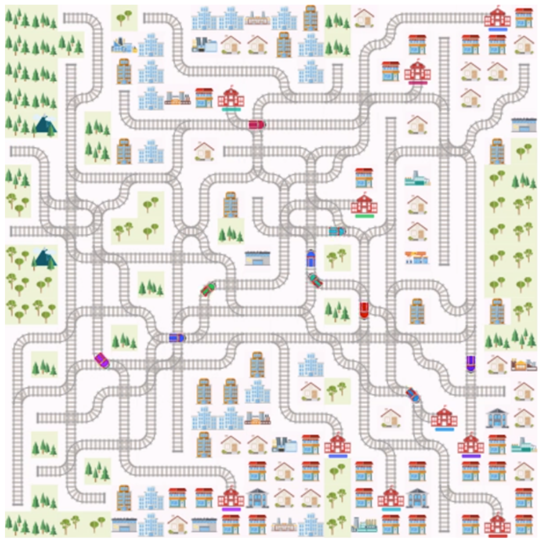
It is important to note that trains do not have the ability to go backwards and therefore need to plan ahead to avoid getting stuck. To learn which actions to take, the agents have to learn to adapt to an unknown environment due to the fact that the environments are randomly generated and differ on each episode. Depending on the given parameters, the size and complexity of the grid can be adjusted. This allows for dynamically changing the difficulty for the agents.
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.

Agent Evaluation

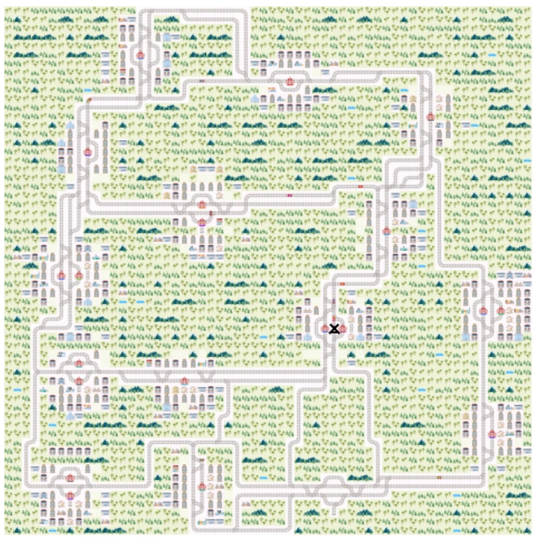
AICrowd and SBB provide a system for agent evaluation. This system evaluates the policy on a number of unknown environments and outputs the percentage of agents that reached their assigned destination as well as the received reward while doing so. The evaluation reward scheme is thereby as follows [11]:

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

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. While round 1 does not have the concept of cities.



(a) Screenshot from Flatland evaluation system for round 1.



(b) Screenshot from Flatland evaluation system for round 2.

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

Observations

The Flatland environment allows to create 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 cells, 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 0* switch heading from north to south is not able to make a decision on this switch and therefore, the TreeObservation 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 to select specific features to create numeric input vectors for function approximator such as neural networks. The tree observation builder offers 14 distinct features for each rail section. This includes:

- Dist. own target encountered: Cell distance to the own target railway station. Inf. if target railway station for agent is not in this section.
- Dist. other agent encountered: Cell distance to the next other agent on this section.
- Dist. to next branch: The length of this section.
- Dist. min to target: The 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).

3. Approach and Methodology

3.1. Iterative Work Approach

As described under section 1.1, our work is based on the work of S. Huschauer [2]. 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 by changing the specified part, followed by an experiment 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 in chapter 5, all technical improvements are also part of our solution for round 2.
Our work can be categorized into a section about reinforcement learning for Flatland (section 4.1) and a section about infrastructure and scaling up the training process (section 4.2).

3.2. Reproducibility and Experiment Specification

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 execute all presented experiments on machines with the same specifications (see section 4.2).
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 4.1.
For better comparability, we try to 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:

- We use an environment of 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 [12].

To analyze the performance of the solution, we run an analyzer that executes 20 rollouts of the environment using the same neural network parameters. After the these 20 rollouts, 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 now take the mean number of agents arrived at their target and plot that in our graph.

4. Experiment Design and Analysis

4.1. Reinforcement Learning for Flatland

A3C Implementation

Originally, the asynchronous advantage actor critic algorithm (A3C) 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 behaviour of the environment changes while training, due to the fact that the behaviour of the other agents changes. Gupta et al.finds in [13], that RL methods like Deep-Q networks (DQN) and Trust region policy optimization (TRPO) are not performing well in a multi agent environment, due to the combination of experience replay and non-stationarity of the environment. We therefore suggest, that it is not recommendable to keep an experience replay buffer with older episodes. Otherwise the sampled experience might represent old agent behaviour which is then learned to deal with.
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 would any earlier update have a reward of 0 and not help the training process.

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 [12]:

> Considering, that the actions and directions an agent can chose 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 which we use to produce an input vector for our neural network. This observation builder takes the 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 fulfil an number of requirements:

- Each rail section the agent possibly rides on next should be visible to the agent.

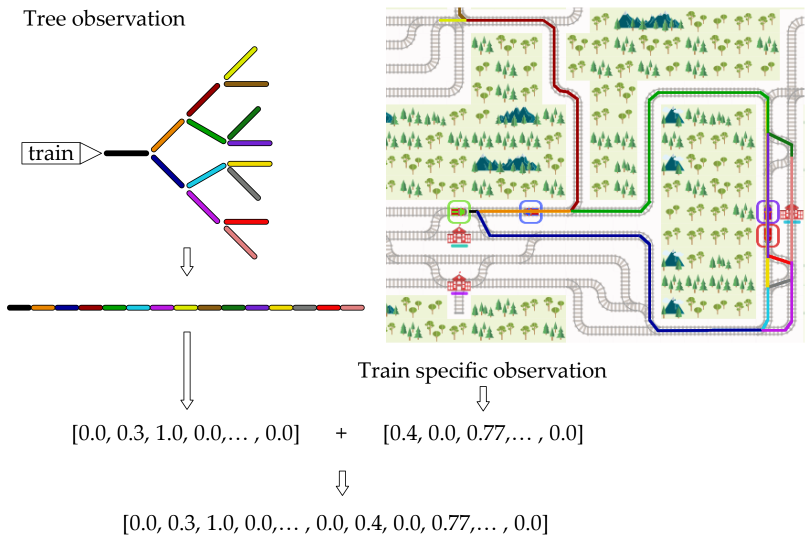


Figure 4.1.: Illustration of the tree observation mapped onto the Flatland enviornment. Each colored bar represents information about a section.

- 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 produce 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.
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

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 pseudocode algorithm 1.

```
Data: initialized Flatland environment $\mathcal{E}$, initial observation $s^{\mathrm{a}}_{\mathrm{t}=0}$ 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 $s^{\mathrm{a}}_{\mathrm{t}}$ of agent
        // Fetch action for agent, based on current state
        $\mathcal{A}[a] \leftarrow$ from policy $\pi(s^{\mathrm{a}}_{\mathrm{t}})$
    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 $\pi$
```

Algorithm 1: Default episode for Flatland environment

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 4.2. Only around switches, an agent needs to take actions other than just keep going forward.

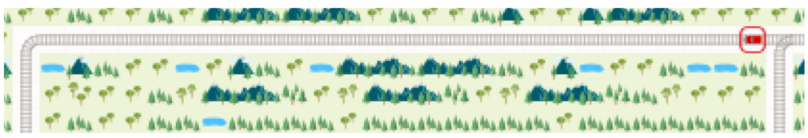


Figure 4.2.: Screenshot from Flatland environment. A train heading east. The only reasonable action is to ride forward.

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 into 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 a large portion of the training data is actually situations that do not actually require decisions.
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 less samples but the samples available are of higher quality. The training with this mechanism implemented looks like algorithm 2. For training, we only use the experience collected near the switch.

```
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 $\pi$
```

Algorithm 2: Improved learning for flatland environment

This drastically improves training performance as visible in Figure 4.3. While both versions have the potential to perform well, based on the observation data, we observe that the version without the action reduction is not able to performe nearly as well as the improved version with reduced actions.

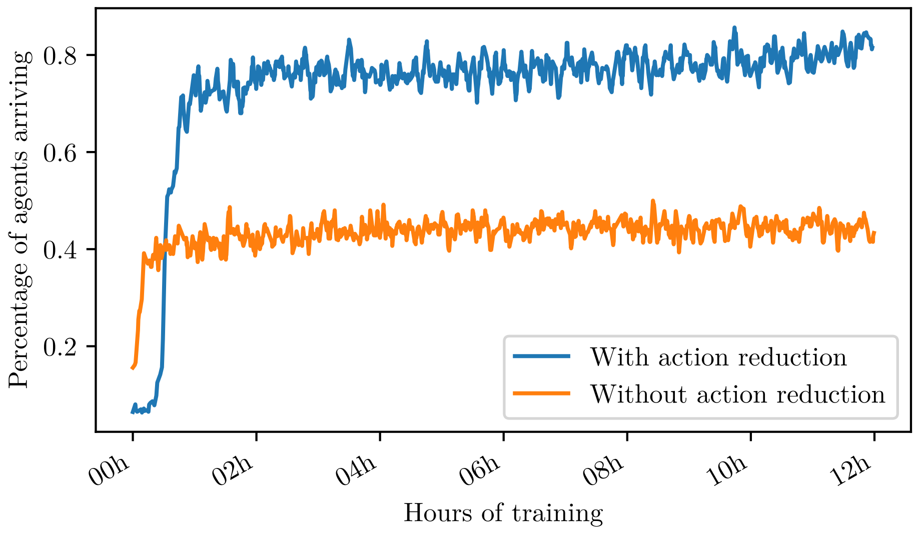


Figure 4.3.: Comparison of training with and without action reduction

As a consequence, we keep the reduced action mechanism as part of our final solution.

Neural Network Architecture

In RL, the architecture of neural networks is often rather simple [6][3]. The already sample inefficient process of reinforcement learning should not be slowed down by a difficult architecture. Differently than S. Huschauer in [2], we do not use a convolutional neural network. The reason for this is the way our observation vector is composed (see section 4.1). 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 more 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. To quantify this improvement, we run an experiment in Figure 4.4 to compare a version with recurrent layer to a version without.

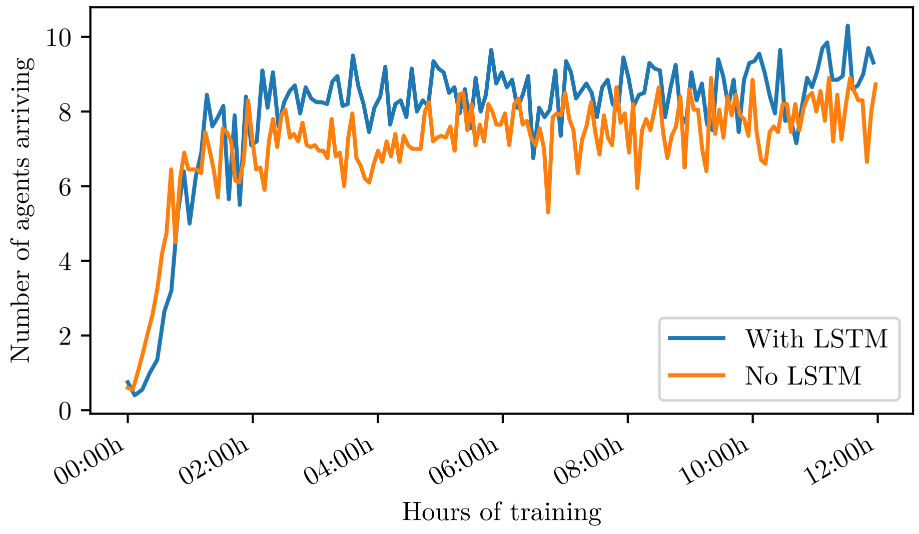


Figure 4.4.: Comparison of training performance between a version with an LSTM-layer and a version without. It is evident that the version with LSTM performs better than the version without.

It is clearly visible that an LSTM-layer helps the training process and we 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 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 (SOURCE!). The rollout of such an environment takes a lot more time than the rollout of a 20x20 environment

	Level 0	**Level 1**	**Level 2**	**Level 3**	**Level 4**
Next level on successrate	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.

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 larget ones, once the agents mastered pathfinding and basic collision avoidance. 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. As visible in Figure 4.5, the version without curriculum is not able to learn from the experience. We suspect that is 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.

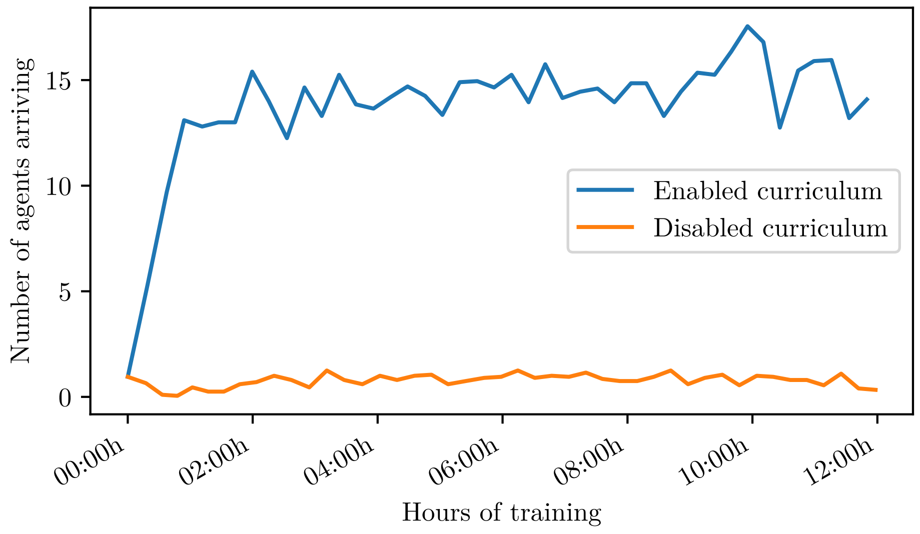


Figure 4.5.: Comparison of training performance between a version with and a version without curriculum learning. It appears, that the version without curriculum is not able to generalize well in larger environments.

While it is clearly visible in Figure 4.5 that directly training on difficult levels is not feasible, it also 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 rollout 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.

Entropy Balancing and Hyperparameter Tuning

In RL, it is of great importance to find the right combination of exploration and explotation [14]. During exploration, the agent explores as much of the state space as possible. This enables the agent to later exploit the found states which are beneficial. Without this exploration phase, there is a chance that the agent settles on suboptimal policies too quickly and ignores parts of the state space the agent has never seen and therefore does not consider.
The policy is characterized by a probability distribution of actions.

Agent Communication

The topic if of communication in multi agent RL is a topic of ongoing research. As a starting point for this work we asked ourselves 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 traffic of all trains on the grid. 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 themselfes. In the following discussion, we summarize these options under the topic of communication and centralized planning. 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 assigend target without communication? We assume that this is not the case. While classical traffic rules (like 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 instances 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 does bring 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, both trains will not be able to arrive at their assigned targets. As a solution to this dilemma, we discuss three solutions that could work to resolve such situations.

- **Negotiation:** To solve the problem of conflicting actions at the same timestep, it would be possible to introduce a iterative communication channel, on which the agents can negotiate, what agent is allowed to take an action next and who has to wait 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 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 artifical 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 policy 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 maybe 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 unconstrainted 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 influcence the actions of the agents. It would be nescessary to learn both the ”speaking” and the ”interpretation”, without a direct consequences.

4.2. 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 to run 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 environements 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 even 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 both the code for building observations 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 into native C-code using Cython. This code converted code then gets dynamically imported.
Using native C-code speeds up the rollout of the environment and therefore the training process by a factor of 2 to 5.

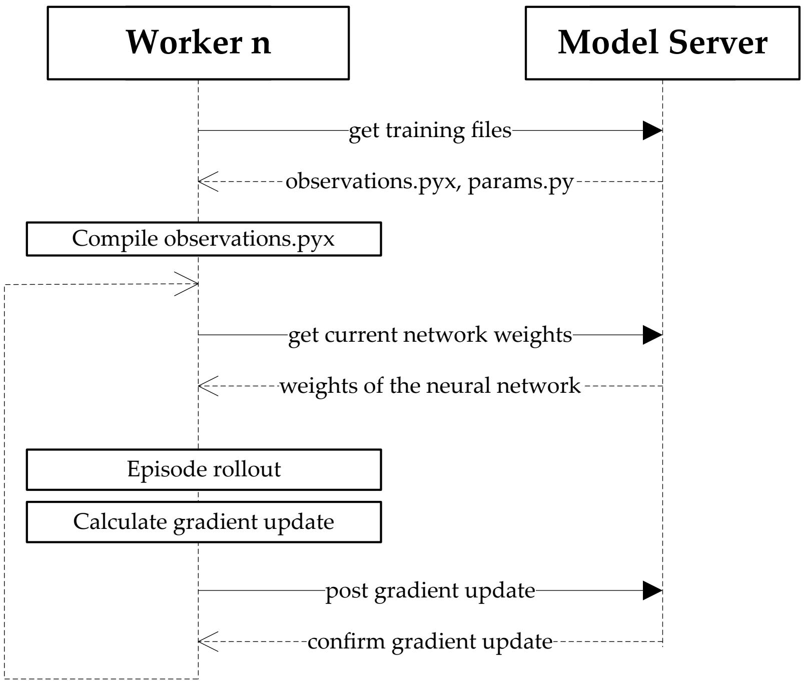


Figure 4.6.: Simplified procedure of training of a single worker process. The communication between the worker and the model server works by using HTTP. The compilation of the observation.pyx file is only done once on each machine.

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

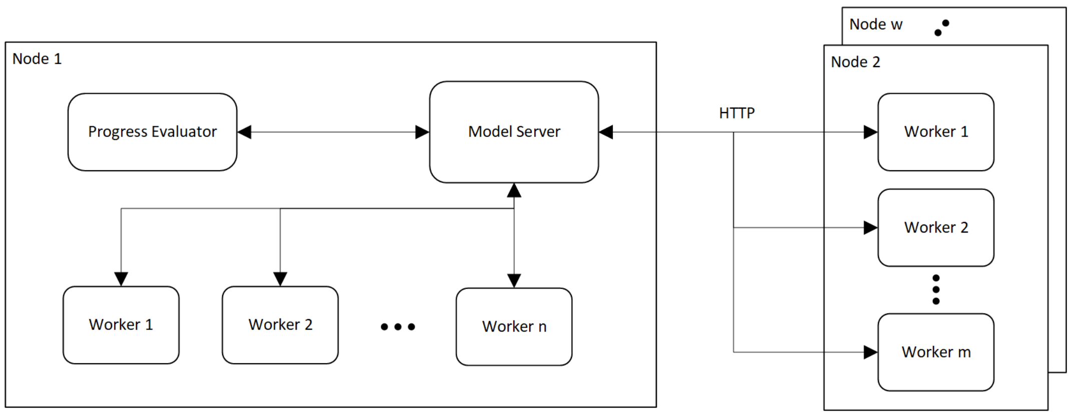


Figure 4.7.: Illustration of distributed architectures with main node (node 1) and several additional nodes (2-w) connected over HTTP.

To analyze the impact of training a policy with multiple workers a the same time, we run an experiment that compares having a single worker to training with 7 workers at the same time. While it is appearant in Figure 4.8 that having mulitple workers does speed up the training process, we do not observe any impact of the number of workers on the quality of the policy.

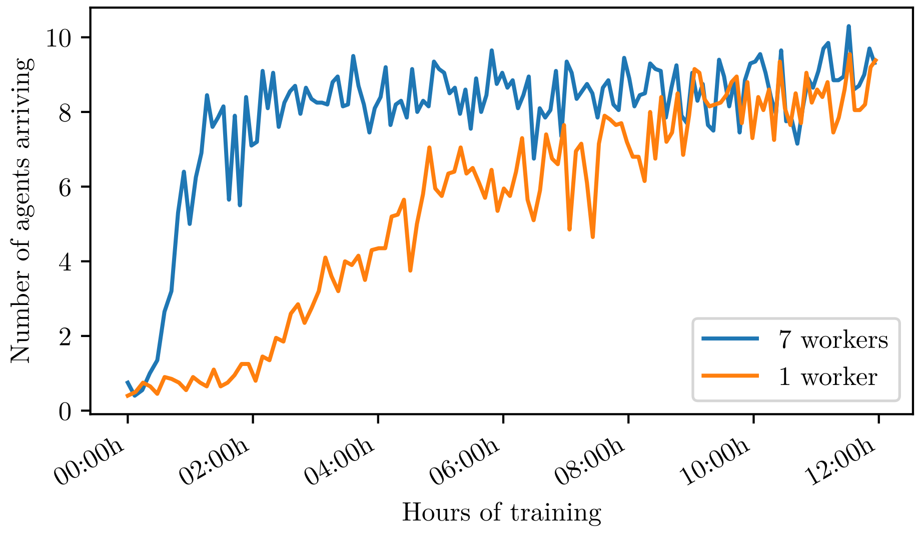


Figure 4.8.: Comparison of training with 1 worker and with 7 workers

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 fragmented update process. Differently than with an experience replay buffer, the training data is only used once and gets then discarded.

- (Beschreibt die Grundüberlegungen der realisierten Lösung (Konstruktion/Entwurf) und die Realisierung als Simulation, als Prototyp oder als Software-Komponente)
- (Definiert Messgrössen, beschreibt Mess- oder Versuchsaufbau, beschreibt und dokumentiert Durchführung der Messungen/Versuche)
- (Experimente)
- (Lösungsweg)
- (Modell)
- (Tests und Validierung)
- (Theoretische Herleitung der Lösung)

4.3. Technical Implementation Aspects

We used the following tools in our project.

Working Environment

- Microsoft Windows 10
- Ubuntu 19.04

Visual Studio Code

- Visual Studio Code 1.40

Documentation

- XeLateX with Visual Studio Code
- XeLateX with WebStorm

Programming language

- Python 3.6

Python modules

- Flatland-rl 1.3 - 2.1.10
- Tensorboard 2.0
- Keras x.x
- Cython x.x
-

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**
Stephan Huschauer	reduced grid observation	19.4%	16.6%
	tree observation	24.7%	-
We	tree observation	69.3%	48.9%

5.2. Round 2

Our submission for Flatland round 2 contains all in this work discussed improvements of the algorithm. 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 algorithm.
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.

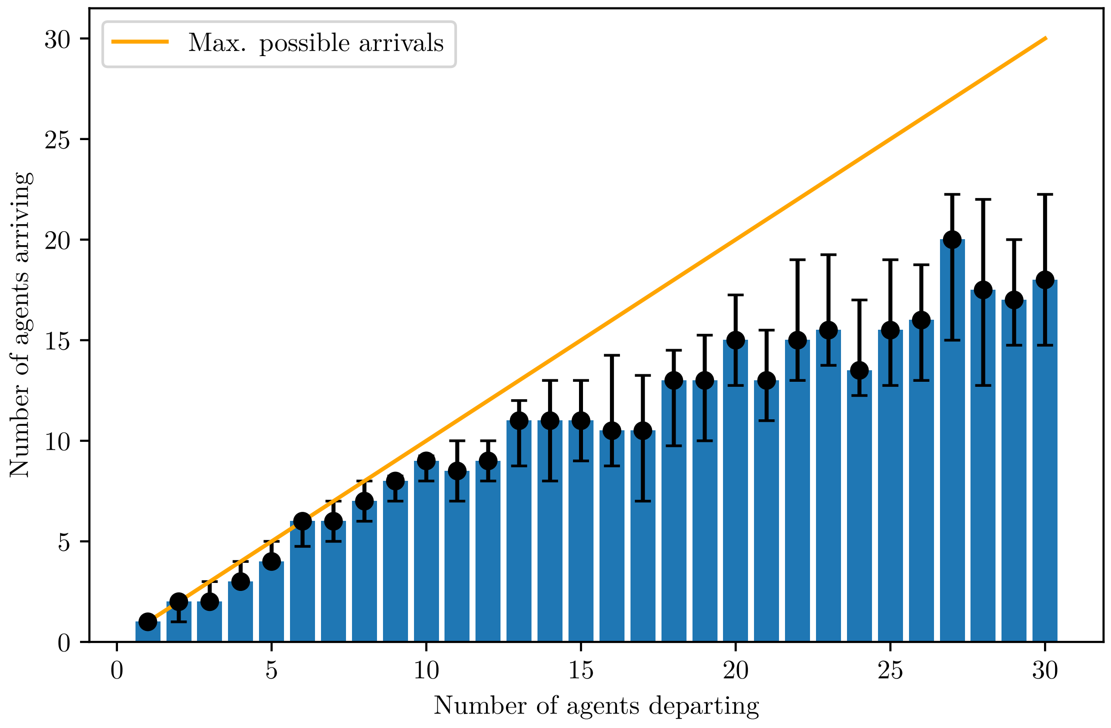


Figure 5.1.: Agents departing vs. agents arriving on a 30x30 environment.

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 absolut number of agents but also on the density of 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.
When analyzing our solution on 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 ares with many switches.

Possible solutions for case 1 have been discussed in section 4.1. We think, that case 2 requires upfront planning, especially in situations with many agents involved. Possiblities to solve situation of this kind are discussed in chapter 6.

6. Diskussion and Outlook

- Bespricht die erzielten Ergebnisse bezüglich ihrer Erwartbarkeit, Aussagekraft und Relevanz
- Interpretation und Validierung der Resultate
- Rückblick auf Aufgabenstellung, erreicht bzw. nicht erreicht
- Legt dar, wie an die Resultate (konkret vom Industriepartner oder weiteren Forschungsarbeiten; allgemein) angeschlossen werden kann; legt dar, welche Chancen die Resultate bieten

7. Verzeichnisse

Bibliography

[1] [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] [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] [Online]. Available: https://gitlab.aicrowd.com/flatland/flatland/blob/master/FAQ_Challenge.md/

[12] [Online]. Available: http://flatland-rl-docs.s3-website.eu-central-1.amazonaws.com/04_specifications.html/

[13] 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.

[14] L. Rusch, “Exploration-exploitation trade-off in deep reinforcement learning.”

List of Figures

2.1. Reinforcement learning overview 8
2.2. Screenshot from a running Flatland environment. 10
2.3. Possible switches in the Flatland environment from [5]. 11
2.4. Comparison between evaluation environments from Flatland round 1 and round 2. 12

4.1. Illustration of the tree observation mapped onto the Flatland enviornment. Each colored bar represents information about a section. 17
4.2. Screenshot from Flatland environment. A train heading east. The only reasonable action is to ride forward. 18
4.3. Comparison of training with and without action reduction 19
4.4. 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. 20
4.5. Comparison of training performance between a version with and a version without curriculum learning. It appears, that the version without curriculum is not able to generalize well in larger environments. 21
4.6. Simplified procedure of training of a single worker process. The communication between the worker and the model server works by using HTTP. The compilation of the observation.pyx file is only done once on each machine. . 24
4.7. Illustration of distributed architectures with main node (node 1) and several additional nodes (2-w) connected over HTTP. 24
4.8. Comparison of training with 1 worker and with 7 workers 25

5.1. Agents departing vs. agents arriving on a 30x30 environment. 28

List of Tables

4.1. Curriculum level specifications for our experiment to compare a version with curriculum to a version without. 21

(Glossar)

In diesem Abschnitt werden Abkürzungen und Begriffe kurz erklärt.

Abk	Abkürzung
XY	Ix Ypsilon
YZ	Ypsilon Zet

Listings

A. Anhang

A.1. Projektmanagement

- Offizielle Aufgabenstellung, Projektauftrag
- (Zeitplan)
- (Besprechungsprotokolle oder Journals)

A.2. Weiteres

- CD mit dem vollständigen Bericht als pdf-File inklusive Film- und Fotomaterial
- (Schaltpläne und Ablaufschemata)
- (Spezifikationen u. Datenblätter der verwendeten Messgeräte und/oder Komponenten)
- (Berechnungen, Messwerte, Simulationsresultate)
- (Stoffdaten)
- (Fehlerrechnungen mit Messunsicherheiten)
- (Grafische Darstellungen, Fotos)
- (Datenträger mit weiteren Daten (z.B. Software-Komponenten) inkl. Verzeichnis der auf diesem Datenträger abgelegten Dateien)
- (Softwarecode)

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.