University of Basel

Deep Reinforcement Learning for Online Planner Portfolios

Bachelor thesis

Faculty of Science
Department of Mathematics and Computer Science
Research Group artificial intelligence

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Patrick Ferber

Tim Goppelsroeder
t.goppelsroeder@stud.unibas.ch
18-057-711

09.02.2023

Abstract

In the field of automated planning and scheduling, a planning task is essentially a state space which can be defined rigorously using one of several different formalisms (e.g. STRIPS, SAS+, PDDL etc.). A planning algorithm tries to determine a sequence of actions that lead to a goal state for a given planning task. In recent years, attempts have been made to group certain planners together into so called planner portfolios, to try and leverage their effectiveness on different specific problem classes. In our project, we create an online planner which in contrast to its offline counterparts, makes use of task specific information when allocating a planner to a task. One idea that has recently gained interest, is to apply machine learning methods to planner portfolios.

In previous work such as Delfi (Katz et al., 2018; Sievers et al., 2019a) supervised learning techniques were used, which made it necessary to train multiple networks to be able to attempt multiple, potentially different, planners for a given task. The reason for this being that, if we used the same network, the output would always be the same, as the input to the network would remain unchanged. In this project we make use of techniques from reinforcement learning such as DQNs (Mnih et al., 2013). Using RL approaches such as DQNs, allows us to extend the input to the network to include information on things, such as which planners were previously attempted and for how long. As a result multiple attempts can be made after only having trained a single network.

Unfortunately the results show that current reinforcement learning agents are, amongst other reasons, too sample inefficient to be able to deliver viable results given the size of the currently available data sets.

Table of Contents

Abstract **ii**

1 Introduction **1**

2 Background **4**

2.1 PDDL Tasks & Image Encoding . . . 4

2.1.1 PDDL . . . 4

2.1.2 Image Encoding . . . 5

2.2 Overview of RL & Methods Used . . . 6

2.2.1 RL and MDPs . . . 6

2.2.2 The Bellman-Equation . . . 7

2.2.3 Q-learning . . . 7

2.2.4 DQN . . . 8

2.2.5 DDPG . . . 9

2.2.6 MARL . . . 11

2.2.7 Reward Shaping & Reward Design . . . 11

2.3 Planner Portfolios . . . 13

2.3.1 Offline Portfolios . . . 13

2.3.2 Online Portfolios . . . 13

3 Reinforcement Learning Agents **14**

3.1 DQN with Constant Action Time . . . 15

3.1.1 Reward Structure: . . . 15

3.2 DDPG Sparse Rewards . . . 16

3.2.1 Reward Structure: . . . 16

3.3 DDPG Reward Shaping . . . 16

3.3.1 Reward Structure: . . . 17

3.4 DDPG Dense Rewards . . . 17

3.4.1 Reward Structure: . . . 17

3.5 MARL Adhere to Region . . . 18

3.5.1 DQN Reward Structure: . . . 18

3.5.2 DDPG Reward Structure: . . . 19

3.6 MARL Adhere to MSE . . . 19

3.6.1 DDPG Reward Structure: . . . 19

4 Experiments **20**
4.1 DQN with Constant Action Time . . . 20
4.2 DDPG Sparse Rewards . . . 21
4.3 DDPG Reward Shaping . . . 22
4.4 DDPG Dense Rewards . . . 22
4.5 MARL Adhere to Region . . . 23
4.6 MARL Adhere to MSE . . . 23
4.7 Comparison between Experiments . . . 24

5 Summary & Future Work **25**

6 Conclusion **28**

Bibliography **29**

Appendix A Appendix **32**
A.1 Network Structures . . . 32
A.1.1 DQN Network Structure: . . . 32
A.1.2 DDPG Sparse Rewards Network Structure . . . 32
A.1.2.1 Actor: . . . 32
A.1.2.2 Critic: . . . 33
A.1.3 DDPG Reward Shaping Network Structure . . . 33
A.1.3.1 Actor: . . . 33
A.1.3.2 Critic: . . . 33
A.1.4 DDPG Dense Rewards Network Structure . . . 34
A.1.4.1 Actor: . . . 34
A.1.4.2 Critic: . . . 34
A.1.5 MARL Adhere to Region Network Structure: . . . 34
A.1.5.1 Actor: . . . 34
A.1.5.2 Critic: . . . 35
A.1.5.3 DQN: . . . 35
A.1.6 MARL Adhere to MSE Network Structure: . . . 35

1 Introduction

Automated planning is an area of AI research that generally concerns itself with finding a sequence of actions for a given planning task, that leads from an initial state to a goal state. An example of a typical planning domain is blocks world. In a blocks world task, the initial state describes how a certain fixed number of distinct blocks lie on a surface, where certain blocks are potentially stacked on top of one another. The actions in the domain correspond to picking up a single block at most, moving between areas and dropping blocks when above a certain area. The goal state or states are determined by restrictions on where, and the configuration in which the blocks should be stacked.

In recent years, it was observed, that no specific planning algorithm excelled across all planning domains. Moreover, it was realized that most planners either solve a task quickly or take a very long time (Helmert, Röger, and Karpas, 2011), which means it is usually more productive to try different planners than to try the same one for a very long time. This motivated the creation of so called planner portfolios, where a planner portfolio is a collection of planners made to leverage the unique characteristics of the individual planners. For solving tasks, we define an upper bound in terms of time available, so that each planner allocated receives a time allocation as well.

Planners in portfolios should ideally complement one another in the sense that they maximize the potential to solve as many tasks as possible (i.e. coverage). Now, given a portfolio with potential for high coverage ,the problem either becomes one of finding a predetermined schedule by statistical analysis of planner performance on a training set of problems, that will be statically used for every problem instance (offline portfolio), or to develop an efficient algorithm that maps a task to a planner and time allocation, while reading in a given state of the problem (online portfolio).

An example of an offline planner portfolio is Fast Downward Stone Soup (Helmert, Röger, and Karpas, 2011), which was trained to develop a sequence (i.e. a schedule) of planners and time allocations for these planners through hill-climbing. Hill climbing refers to a local search algorithm, which constantly moves in the direction of increasing values to find the top

of the hill (i.e. the optimal solution). In each hill-climbing step, Stone Soup generates all possible successors of the current portfolio. There is one successor for each planner, where the only difference between the current portfolio and the successor is that the time limit of a certain planner is increased by a given granularity. The quality of a portfolio is computed by calculating its portfolio score. The portfolio score is the sum of all the instance scores. If there is no planner in a portfolio that solves an instance within its allotted time, the instance score is set to zero. If this is not the case, the instance score is set to a ratio between the lowest solution cost and the best solution cost. In each hill-climbing step the successor is chosen with the highest portfolio score. The hill-climbing phase terminates at the point when all successors would potentially exceed the given time bound. A post-processing step sets the planners time limits to the lowest value that would still preserve the portfolio score. One disadvantage of offline planners such as Stone Soup is the fact that the decision of what the planner and time allocation should be, is task independent. This means that the algorithm doesn't make use of task specific information, such as the planners already attempted for a task, or the time remaining for a task.

An example of online portfolios is Delfi (Katz et al., 2018; Sievers et al., 2019a), where planning tasks were encoded as images, which allows the use of CNNs (which are comparably well understood and developed) to identify patterns from the images. In Delfi, a neural network was trained, using experimentally predetermined runtimes for each possible pairing of a planner and a task, to predict a single planner, for a given image representation of a task. Further work by Ma et al. (2020) used techniques that have recently seen advancement, such as the use of graph neural networks (GNNs), which avoid losing information when encoding the task as an image, to improve results and made two attempts with half the total time per task. Due to the use of supervised learning in this project, it is necessary to train multiple networks (n networks) to make multiple attempts (n potential attempts), as neural networks are deterministic after training and the input to the network would remain the same. An attempt consists of, the trained network, predicting a planner and a time allocation for the current task's current state. Unfortunately, this approach does not scale well to multiple attempts as was also described in the paper Online Planner Selection with Graph Neural Networks and Adaptive Scheduling (Ma et al., 2020). In essence, the number of networks that would have to be trained would scale linearly with the number of attempts we want to make, and for every new network we train, the training process would take longer, since we are training the network on all possible intermediary task states, that the previous network reached.

A few benefits of online portfolios are that making multiple attempts per task generally increases the potential that one of the planners can solve the task within its allocated time, and that a less uniform time allocation for the attempts made is likely to increase efficiency and therefore the number of attempts that can be made per task, as for example certain planners are known to either solve the task in a short amount of time or not at all. Some disadvantages of online portfolios are that they are generally more difficult to train than their offline counterparts, and require additional computational overhead after training by

evaluating which planners to use for each task, instead of adhering to a predetermined schedule.

In this project, we would like to extend this idea of making multiple attempts to be able to make any number of attempts per task using a single network. We achieve this by exchanging the previously used supervised learning methods for reinforcement learning (RL) methods. These RL methods allow us to formulate the planner portfolio problem as a "game", where we can pass additional information aside from the image representing the task such as a history of previous planner executions, the amount of time the most recent planner ran for, and the time left to solve the task. The agent is rewarded for the desired behavior (i.e getting as many tasks correct within their time limit). The "additional state information" should give the network a chance to guess a different planner when asked about the same task repeatedly, since the input to the network will not be the same. This would then solve the scaling problem of having to train multiple networks and allows us to make multiple attempts for a single planning task, despite only having trained a single network.
We begin by training a deep Q-network (DQN) to select planners with a constant fraction of the time, where the fraction is a hyper-parameter that determines the amount of attempts made. The reason for starting with DQN is the fact that it is easier to implement than other RL algorithms and because this allows us to view the environment as a simplified discrete version. By additionally dynamically allocating a time output for each planner for each task state, instead of just allocating a constant fraction of the total time we make the action space continuous. To be completely accurate, the action space is now a hybrid action space that is partially continuous and partially discrete. DQN is known to not be able to deal with continuous action spaces well, thus we turn to actor-critic methods specifically deep deterministic policy gradients (DDPG), which can handle continuous action spaces. Three different versions of DDPG with different reward functions are explored. The first having a sparse rewards function. The second made use of reward shaping to incentivize intermediary goals. The last version rewarded the whole output planner vector from the DDPG agent and represents the densest reward function for any of our DDPGs. Finally, due to the action space being a hybrid action space, multi-agent reinforcement learning (MARL) was used, as MARL has shown more promising results in dealing with hybrid action spaces than pure actor-critic methods. In terms of MARL agents, two additional experiments, both separately using a DDPG for time allocation and a DQN for planner allocation, while using different reward functions, were created to explore this idea for planner portfolios.

2
Background

2.1 PDDL Tasks & Image Encoding

In planning, tasks are viewed as state spaces where the objective is to find a sequence of actions leading from some initial state to a goal state. In optimal planning, which is what we use in this project, the objective of a planner is to find such a sequence with minimal cost. There are many different planning formalisms, such as $PDDL$, SAS^{+} and $STRIPS$. In this project, tasks are encoded using a formalism called $PDDL$. Below, we define a simplified version of $PDDL$ without conditional effects, action costs or axioms.

2.1.1 PDDL

PDDL (Ghallab et al., 1998) is a formalism that describes tasks using first-order logic. A PDDL task consists of objects which are essentially entities in the environment that are of interest to us and predicates. Predicates have objects as arguments and describe statements that hold in a state, where the union of all such statements describes the state. A PDDL task also consists of an initial state, a goal formula and finally actions that can change the current state s by transitioning to a new one s' on an arbitrary action a.

Definition 2.1.1. *PDDL Task* $\Pi = (\mathcal{O}, \mathcal{P}, \mathcal{A}, s_1, \delta)$

- *$\mathcal{O}$ is a finite set of mathematical objects that represents the environment*
- *$\mathcal{P}$ is a finite set of predicate symbols with each predicate symbol having arity $k(p)$, where $p \in \mathcal{P}$. A predicate is a predicate symbol associated with a $k(p)$ tuple that can be made up of objects and variables. If the predicate is only composed of objects, then it is considered grounded (grounded predicates are also known as facts). If there is at least one variable, then it is considered lifted. A predicate can be grounded by assigning values to all of its variables.*
  F is the set of all facts and $F_\neg$ the set of all negated facts where F and $F_\neg$ are independent. Also the set of all literals is defined as $\mathcal{L} = F \cup F_\neg$. Given these definitions, a state in the current environment can be defined as $s \subseteq F$. s can be extended to include the remaining facts as negative literals $[[s]] = s \cup \{\neg f | f \in F \wedge f \notin s\}$.

- *$\mathcal{A}$ is a finite set of action schemas, where an action schema $a \in \mathcal{A}$ can be described by a tuple (V_a, pre_a, eff_a). Here, V_a is a set of variables, the pre_a is a set of predicates called precondition of the action and eff_a is a set of predicates that are not necessarily grounded. With a variable assignment V for V_a, pre_a can be grounded to $pre_{a,V}$ and eff_a grounded to $eff_{a,V}$. A grounded action is an action schema, which is grounded with a variable assignment V_a over $\mathcal{O}$. The grounded action a_V, is applicable in a state $s \in S$, written as applicable(s, a_V), if $pre_a, V \subseteq [[s]]$. Given that a_V is applicable in s, then applying it, written as $s[[a_V]]$, leads to a new state determined by the effects of the action schema.*
- *s_1 is the initial state*
- *$\delta \subseteq \mathcal{L}$ is a set of literals called the goal. A state s is a goal state, if $s \supseteq \delta$ and S_F is the full set of all goal/final states.*

2.1.2 Image Encoding

The images used in this project were developed using the same techniques that were used in the Delfi paper (Katz et al., 2018). This consisted of formulating the task at hand as an abstract structure graph. The problem/task is first brought into graph form (abstract structure graphs (Sievers et al., 2017)). We can then extract the adjacency matrices from these graphs, which contain important information about the task, and interpret them as grey-scale images. These images are then used to train our networks.

Definition 2.1.2. *Abstract Structure (Sievers et al., 2019b)*
Given a set of symbol types T, a set of symbols S and a function $t : S \mapsto T$, which associates every symbol with a type, we can define an abstract structure as follows:

- *A symbol $s \in S$ is an abstract structure.*
- *A tuple $A = (A_1, ..., A_n)$ of abstract structures A_i is an abstract structure.*
- *A set $A = \{A_1, ..., A_n\}$ of abstract structures A_i is an abstract structure.*

Definition 2.1.3. *Abstract Structure Graph (Sievers et al., 2019b)*
Given an abstract structure A over a set of symbols S that have symbol types T and a function $t : S \mapsto T$. Let $G = (V, E)$ denote its abstract structure digraph, where we define the following:

- *If $A \in S$, then V contains a single node n_A for the symbol A and E is empty.*
- *If $A = (A_1, ..., A_n)$, then V contains a main node n_A. Furthermore, G contains the nodes and edges for the ASG (abstract structure graph) of $A_1, A_2, ..., A_n$ and an auxiliary node $n'_{A,i}$ for every structure A_i . Let $n_{A,i}$ be the main node for the ASG of A_i .E contains edges $n'_{A,i} \mapsto n_{A,i}$. Finally, E contains an edge $n_A \mapsto n'_{A,1}$ and for $1 \leq i < n$ an edge $n'_{A,i} \mapsto n'_{A,i+1}$.*

- *If* $A = \{A_1, ..., A_n\}$*, then* V *contains a main node* n_A*. Furthermore,* G *contains the nodes and edges for the ASG of* $A_1, A_2, ..., A_n$*. Let* $n_{A,i}$ *be the main nodes for the ASG of* A_i*. For every* A_i*, there is an edge* $n_A \mapsto n_{A,i}$

2.2 Overview of RL & Methods Used

Reinforcement learning (RL) is a machine learning technique that trains agents to make decisions, in an environment, that maximize the return (i.e. total discounted reward). Through trial and error, RL algorithms constantly attempt to improve their decision-making. In RL, an agent executes actions in an environment and obtains feedback in the form of rewards. This feedback is analyzed by the agent to refine its comprehension of the environment and improve its decision-making policy. This cycle continues, until the agent either attains a sufficient level of proficiency, at which point it can be used to make real-world decisions, or the algorithm converges before reaching this level of proficiency, in which case the agent's decision making policy will be sub-optimal.

2.2.1 RL and MDPs

One of the substantial differences between supervised learning and RL is that in RL, we have an environment where we estimate the value (i.e reward) of an action given a state, rather than using a large data set that tells us what these values should be, from which the agent hopefully derives patterns that are relevant, when acting on new unseen data. The environment has methods to reset the environment and step through environment. The step function is given an action as an argument and returns a specific successor state, that depends on the input state, before the action was taken. The entirety of this process is called a transition and can be formally defined as a tuple such that $\mathcal{T}^{'} = (S, a, S')$, where S is the original state, a is the action and S' is the successor state. In RL, there is an additional value returned for every transition, called the reward $\mathcal{T}_{RL} = (S, a, S', R)$, where R is the reward and is dependant on $T^{'}$. The reward indicates how good the agent is performing in the environment while training and is used to estimate optimal values by way of the Bellman equation, and thus direct the network in an advantageous manner towards better actions. The reward can also be used to measure how well the agent performs after training on the test set, which was done in the DQN Atari paper (Mnih et al., 2013). However, the reward value can easily be exchanged for a metric, such as percentage correct at this point. The network will try to find patterns in the data that will allow it to emulate these estimated values better by updating its weights and biases. In short, the network is trying to maximize the cumulative reward (a.k.a return) throughout the training instances, which are called episodes. In RL, a state is a collection of information on the environment at a specific moment that is "complete" in the sense, that the information on the state is adequate to ascertain the future states of the environment given any action that is taken in that state. If the following is true for all the states in our state space, then we say that these states obey the Markov property and we can formalize the state space as a Markov decision process or MDP for short (Bellman, 1957b). A Markov decision process is formalized as follows

$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ where $\mathcal{S}$ is the full set of markovian states, $\mathcal{A}$ is the full set of possible actions that can be taken in any of the states in $\mathcal{S}$, $\mathcal{P}$ is a transition function that determines the probability of transitioning from a given state to any of the states that are reachable from the given state and $\mathcal{R}$ is the reward function that determines the agents reward given a transition. In reinforcement learning, the only thing that our network is allowed to assume in general is that it is part of a MDP. The goal of most RL agents is to maximize the return in the given MDP.

2.2.2 The Bellman-Equation

The Bellman equation is a key concept that arose out of progress made in dynamic programming. Dynamic programming is an optimization technique that splits problems into smaller sub-problems (i.e. divide and conquer) to help solve them. The idea was first proposed by Bellman (1952).
In its essence, the Bellman equation expresses the relationship between the state of a problem at a given time and the state of its sub-problems and can be derived from Bellman's principle of optimality (Bellman, 1957a). It defines the value of a state as the maximum expected return that can be obtained by taking an action and transitioning to a new state.

Mathematically, the Bellman equation can be defined as follows:

$$V(s) = max\{R(s,a) + \gamma V(s')\}$$

Where, $V(s)$ is the value of the current state s, R represents the current reward received for transitioning to the new state s' on action a, and $\gamma \in (0,1)$ is a discount factor that makes sure future rewards account for less of the value of the state than the current rewards. This is because if a problem has an infinite time horizon, the objective is to maximize the sum of discounted rewards (MDP with an infinite horizon discounted reward criteria), and iff $|\gamma| \notin (0,1)$ the sum would not converge to a sensible value that takes future rewards into account. Specifically, in the case where $|\gamma| \geq 1$ and the average reward is non-zero, it would result in a positive or negative infinite sum.

The Bellman equation is used to determine an optimal policy for solving a task and can even guarantee an optimal policy given that the number of attempts is infinite and the algorithm stores the exact values for all individual states. This policy corresponds to an action sequence $(a_0, a_1, a_2, ..., a_{n-1}, a_n)$ that maximizes expected return, which is accomplished by sequentially selecting the maximum value action out of all possible actions that can be taken from the current state.

2.2.3 Q-learning

Q-learning is an algorithm that was first introduced by in 1989 (Watkins and Dayan, 1992), and is a very important precursor to the later developed deep Q-networks, which were developed by Deep-Mind in 2013 (Mnih et al., 2013). In Q-learning we create and update a

so called Q-table. This Q-table is a reference table for our agent to pick the optimal action based on the Q-value. It has Q-values for each state action tuple, where a Q-value is a value that estimates the quality of a certain action being taken in a given state. In RL, there is a important trade off between exploration and exploitation. Originally, the Q-table is initialized with some constant values, so that it is unlikely to deliver accurate predictions. As a result, only exploiting the Q-table and its values seems to be an extremely inaccurate approach. The opposite would be to completely ignore the Q-table and just take random decisions (i.e. to explore). However, only taking random decisions is trivially unlikely to lead to optimal or even desirable results in almost all cases. Thus, we need a balance between exploration and exploitation in RL, where exploring ideally helps increase the quality of estimates made by our Q-table, so that exploitation of this Q-table leads to more accurate results. The degree to which the agent explores/exploits, is usually determined by a constant (e.g. epsilon) smaller than 1, that indicates the probability of exploring the environment instead of exploiting the Q-table. In certain other variants, the probability that the agent will explore can be made to decay over time (ϵ-decay), so that the agent tends to explore less as time goes on, because it is assumed that the agent is becoming more accurate through its exploration of the environment. In other variants, a noise parameter vector (usually Gaussian noise) is added before selecting the maximum Q-value.

2.2.4 DQN

DQN is a variant of Q-learning where the Q-table is replaced by a neural network (policy network for actions and a target network that is used to estimate future rewards) and was first proposed in 2013 by researchers at DeepMind (Mnih et al., 2013) to deal with high-dimensional sensory inputs (e.g. computer vision). DQN tends to be implemented using ϵ-decay. Below the pseudo-code for DQN from the paper "Playing Atari with Deep Reinforcement Learning".

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

The output layer of the network usually uses a softmax activation function, which allows us to interpret the output layer, as the Q-values associated with taking specific actions in

the current state. The action with the maximum Q-value is selected, a reward is given based on the current state-action tuple and the whole transition (including the successor state) is stored in a so-called Replay memory. A Replay memory stores previously observed transitions, so that we can sample mini-batches uniformly from the Replay memory when training, and update our network based on the expected values that we calculate using the Bellman equation, where Q^* is the policy network, Q is the target network and γ is the discounted proportion of future rewards. However, if the next state is a final state, then the expected value is simply the reward received for the current transition. One of the key reasons for the introduction of a Replay memory, was to break the correlation between consecutive samples, which leads to more efficient learning. DQN is known as an off-policy learner, meaning Q^* approximates the optimal action-value function Q^+ independent of the policy being followed. DQN is usually implemented with a hard update of the target network every x episodes, meaning that the target network copies the weights and biases from the policy network. A target network helps maintain stability when learning and stops the policy network from "spiraling around", because we are allowing the weights to move consistently towards a target before the hard update and the optimization problem is changed. DQN calculates the MSE with respect to the expected values (i.e. the loss) and takes a gradient descent step based on this to update the network.
Plain vanilla DQNs are famously unstable and inefficient learners, even when compared to other algorithms in RL (Irpan, 2018). For example, the Atari paper by DeepMind (Mnih et al., 2013) was a decisive win for deep RL as a whole and spurred massive development in the area, but in most cases a DQN can simply be outperformed by an off-the-shelf Monte Carlo Tree Search (MCTS). Another flaw of DQN is its inability to handle continuous action spaces. Nevertheless, in certain cases it is possible to discretize the continuous action space, so that a DQN can be made to operate on it. Although by now, RL offers better ways of dealing with this that tend to be more stable and are more likely to lead to desired results, such as policy gradient methods and actor-critic networks.

2.2.5 DDPG

DDPG (Lillicrap et al., 2015) is an adapted version of the original DQN that is able to handle continuous action spaces, and does not work particularly well when given a discrete action space. One key change between the DQN network structure and the DDPG network structure is that the DDPG is a so called actor-critic network. In an actor-critic network, the network is separated into two parts, the actor and the critic. The Actor μ is responsible for estimating an action a given the current state s in the continuous action-space. The Critic Q is responsible for evaluating the quality of the state action pair (s, a) and returns a Q-value that is again used to update the Bellman-equation like in DQN. The Q-value is calculated as the discounted reward of the action taken at some state (s, a). Another difference is the use of random action noise (due to continuous action space). In the original DDPG paper, an Ornstein-Uhlenbeck process was used to generate noise. However, this seems to be excessive in most practical applications and Gaussian-noise is preferred in most current implementations, as it is simpler and seems to be just as effective on average. The noise

is added to the value/returned by the actor to promote exploration, instead of just taking random actions with a certain probability ϵ, like it was done in the plain vanilla DQN. There are also many similarities between the two, such as the use of a Replay memory to break the correlation between consecutive samples and both being off-policy learners. DDPG also makes use of target networks, however, unlike DQN, it needs two target networks: one for the actor and one for the critic. This means that a plain-vanilla implementation of DDPG has a total of four separate networks. DDPG also updates these target networks, but does so more often and uses a so called soft-update, which is usually applied after every epsiode instead of after every x episodes. A soft-update of a target network Q with a policy network Q^* is defined as:

- $weights(Q_{new}) = weights(Q^*) * p + weights(Q) * (1 - p)$
- $bias(Q_{new}) = bias(Q^*) * p + biass(Q) * (1 - p)$

Where $p \in (0, 1)$ denotes to which degree we favour the weights/bias from the policy networks each time that we update our target networks. Note that if $p = 1$ it would give the same effect as a hard update.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$$

  end for
end for

In DDPG, the Critic is updated in a similiar manner to how we updated the actor in DQN, where we sample a batch and use the MSE with respect to the expected value to calculate the loss. However, the Actor now needs to be updated with respect to the Critic (policy update). This is done in practice, by having the actor network take actions on the samples from the batch. Then the critic network evaluates these state action pairs to create Q-values, of which the average is taken. The negation of this obtained average is then used to take

a gradient descent step in the actor network with respect to its parameters, as this is the easiest way to do gradient ascent in most machine learning frameworks.

2.2.6 MARL

Multi-agent reinforcement learning, or MARL for short, is a promising sub-field in RL that studies the behavior of agents that interact in a shared environment.
MARL can be further separated into two sub-fields, namely adversarial MARL and cooperative MARL. In adversarial MARL, agents are pitted against each other in a competitive environment. An example of this was the OpenAI paper that trained agents to play hide and seek (Baker et al., 2019). In this paper, there were two kinds of agents the hiders and the seekers and like in many competitive multi-agent environments, it was observed that the agents profited from a competing agent by oscillating between adaptation to the new policy of the competing agent and exploiting dominance over the competing agents policy. In RL, competitive MARL is known to be quite stable when compared to most other RL methods. In Cooperative MARL, the agents will try to interact symbiotically to achieve a common goal. Cooperative MARL has recently shown alot of promise at solving tasks with hybrid action spaces. A hybrid action space is an action space that is neither completely discrete or continuous. An example of a hybrid action space would be the action space defining movement for a humanoid robot. The discrete component of this action space is the choice of which limb or combination of limbs to move, whereas the continuous component indicates how the selected limb should be moved. This method shows alot of promise, but is still outperformed by more traditional optimization and machine-learning techniques at the moment. In our project, cooperative MARL constitutes having a DQN to estimate the discrete component of the action a_d given a state s, and a DDPG that is given this discrete component and determines the time for that action a_t, thus $a = (a_d, a_t)$. How and what agents communicate with one another is paramount to the success of a cooperative MARL model. There are many approaches to modeling interactions between agents in cooperative MARL systems. A few of the main ideas are outlined below:

- agents learn independently of one another
- consensus (using sparse communication)
- communication (agents don't just learn the policy with respect to the task but also learn with whom and what to communicate)

For the interaction between our DQN and DDPG, we opted for a basic form of consensus with sparse communication, where the DDPG is given the DQN's output as an additional part of its input.

2.2.7 Reward Shaping & Reward Design

In RL, a reward is a metric for the quality of an action $a \in A$, given a state $s \in S$, where the reward can be formalized as a mapping $r : S \times A \mapsto R$, where R is the set of possible reward values. The purpose of a reward is to provide some measurement of progress relative

to the goal and thus guide the decision making process.
The simplest forms of reward are so called sparse rewards. A sparse reward in an environment would amount to giving a reward of $+k$ iff the action taken in the current state leads to a goal state, $-k$ iff the action leads to a non goal terminal state, and 0 iff the action leads to a non-terminal state where $k \in \mathbb{R}$. Varying k has a limited influence on the network's learning rate and is commonly set to $k = 1$. Some frequent problems for sparse reward functions include the slow rate at which they learn due to the minimal amount of information received through the reward function, and the potential for falling into local optima.
Reward shaping generally refers to engineering a reward function that gives more frequent feedback than a sparse reward function (i.e. attempts to make the reward function more dense and informative). Reward-shaping usually speeds up learning due to the more frequent feedback. However, we run the risk of falsely assuming that certain intermediary behaviors are beneficial towards reaching our end-goal and may therefore reward them inappropriately, which will likely lead to undesirable results.
A lot of consideration has to be put into reward function design, as any flaw in the reward can potentially lead to reward hacking. Reward hacking refers to an agent finding ways to maximize its reward in ways that were not intended by the programmer, when writing the reward function.
Local Optima are an even larger problem when designing reward functions than with sparse rewards, especially in complex high dimensional environments. For example, A. Irpan (Irpan, 2018) provides an instance of an RL agent that found an interesting local optima in the HalfCheetah environment, where the agent learns to traverse a plane by moving forward. However, the agent learned to flip onto its back before using the convulsions of its back muscles to propel itself forward towards the goal (this is clearly not was originally intended).
Reward-shaping can alleviate this problem if done correctly, but may also exacerbate the problem, if the reward function incentivizes sub-optimal behavior.

2.3 Planner Portfolios

Planner portfolios are collections of planning algorithms designed to complement each other and maximize the potential that a planner in the portfolio can solve a given task. Two distinct types of planner portfolio exist, as outlined below.

2.3.1 Offline Portfolios

Definition 2.3.1. *Let P be a set of planners, $\hat{t}$ an upper bound on time to be used per task, $\mathbb{N}_k = \{n|n \in \mathbb{N} \wedge n \leq k\}$ and $k \in \mathbb{N}$. Then an offline planner can be defined as a function $f(n_i) = (p_i, t_i)$ where $p_i \in P$, $t_i \in \mathbb{R}_+$, $n_i \in \mathbb{N}_k$ and $\Sigma_{i=1}^{k} t_i \leq \hat{t}$.*

In offline portfolios, the model is trained on a fixed size batch of data and the output of the model is a planner schedule, as defined above. At later time intervals the model can be updated using new batches of data if necessary and more data exists.
An example of an offline planner portfolio is Stone Soup (Helmert, Röger, and Karpas, 2011).

2.3.2 Online Portfolios

Definition 2.3.2. *Let P be a set of planners and $\hat{t}$ an upper bound on time to be used per task. Given a task Π and some information on previously attempted planners and time left(originally $\hat{t}$ for every task) that we call a history H, an online portfolio can be formalized as a function $f : (\Pi, H) \mapsto (p, t)$ which predicts the next planner $p \in P$ and a time allocation $t \in \mathbb{R}_+$ for this planner to execute for the task Π given history H.*

In online portfolios, the model is trained incrementally, using a constant and up to date stream of data, that describes the current state of the current planning task. The model is updated with each new piece of data and is able to make predictions based on task specific information.
Our planner portfolio is given a task encoded as an image to operate on for a given episode, as well as additional information about the task such as a history of planners that have already been attempted.
An example of an online planner portfolio is Delfi (Katz et al., 2018), which used supervised learning techniques and partially inspired the idea behind this project.

3 Reinforcement Learning Agents

The task data we use for our experiments is the same data that was created and used for the training of Delfi (Sievers et al., 2019a). The planner set for our portfolio consists of the same planners as Delfi (Katz et al., 2018), which participated in the IPC 2018. It includes 17 Fast Downward based planners. The planners include a DKS structural symmetry pruning (Domshlak, Katz, and Shleyfman, 2012) and a orbital space search structural symmetry pruning (Domshlak, Katz, and Shleyfman, 2015) variant. Each of these variants using one of the following eight heuristics: blind heuristic, LM-Cut (Helmert and Domshlak, 2009), iPDB (Haslum et al., 2007) with a time limit of 900 seconds for the pattern generation, a zero-one cost partitioning pattern database (ZO-PDB) which generates the patterns using a genetic algorithm (Edelkamp, 2006), and four Merge-And-Shrink (M&S) heuristics (Dräger, Finkbeiner, and Podelski, 2006; Helmert et al., 2014). As well as the planner $SymBA^{\star}$ (Torralba et al., 2017). Also for a precise definition of the network structures used in the individual experiments refer to the appendix of this paper.

With this data we train multiple agents to allocate planners to tasks, where in certain cases the agent also allocates a time slot and in others this slot is constant. The agent samples a task from our data set and makes repeated attempts (i.e. a planner allocation for a certain time period that is either fixed or comes from the network) until the time for the episode is done. For every attempt the agent has an input comprised of an image encoding of the task and additional information consisting of: the previously attempted planners and their previous consecutive runtimes, the most recently run planners runtime and the time left in the current episode. As output we receive a planner choice and if the agent does not use a fixed time allocation, we also receive a time allocation. To balance exploration and exploitation the agent ignores its policy and takes a random action with a certain probability epsilon (in some other variants this is implemented using Gaussian noise). After transitioning to a new state based on taking a certain action, a reward function tells the agent how qualitatively good the attempt was and is used to calculate an expected value based on the reward value and the estimated Q-value of the state by the network. Using this expected value we calculate the MSE for a batch of such transitions to the obtained value and take a gradient descent step based on this in the network.

For every agent, we test the networks on a test set after every 10 episodes of training. The test set is independent from the training set in the sense that there is no task overlap and that the split is also domain independent (i.e. all tasks of a certain domain belong to the same set). Also, time outputs are clipped to be at most the time remaining, unless otherwise specified and for all DDPG variants gradient clipping is used, which can help prevent exploding and/or vanishing gradients.

After testing the networks, we calculate the average reward obtained during the testing process. If the average reward is larger than any of the average rewards obtained previously while testing, we save the network configuration in a separate file (store network configuration for potential use later when compiling results and creating visualizations). A relatively small batch size of 32 was used to increase generalization (Keskar et al., 2016) and to reduce compute time per episode for all of our experiments.

3.1 DQN with Constant Action Time

In previous work on online planners supervised learning was used to make a fixed number of attempts per task. However this required training multiple networks to do this and did not scale well. The simplest extension using deep RL that we could think of was to discretize the environment, by making the time allocation constant, and using a DQN to predict which planners to use for tasks, thus allowing us to be able to make multiple predictions, after only having trained one network. In the DQN implementation, the time output was kept constant where $a_t = 180$ or 10% of the total time. A DQN outputs a planner selection, which is paired with the constant time output in a tuple to give the full action. In our DQN, the exploration/exploitation trade-off was modeled using epsilon decay, where the starting epsilon is 0.9 and is slowly reduced to 0.05 using a decay factor of 200. The target network is updated using a hard update every 20 episodes.

3.1.1 Reward Structure:

The reward structure for the DQN is simple. We give a scaled positive reward if the planner solves the task and 0 for all planners that cant solve the task.
The reward is determined by ordering the planners that would solve the current task in the remaining time from best to worst. We give 1 for the best planner, 0 for all the planners that cant solve the task in the remaining time and for the remaining planners, we interpolate between 0.5 and 1 based on the position of the planner in the sorted solving planners. The variable Done is also returned from the reward function and indicates if the planner selected can potentially solve the task in the remaining time (environment then checks if time allocated is sufficient to determine if the next state is a final state or not).

3.2 DDPG Sparse Rewards

After the somewhat limited results shown by the DQN attempt we felt it seemed logical to create an agent that operates on the original action space, with its continuous time component, instead of the simplified, discretized version. To be able to handle continuous action spaces we switched to actor-critic networks specifically DDPG. A sparse rewards approach was used here to make sure that we are not unnecessarily biasing the reward function by for instance applying incorrect reward-shaping techniques. This can, however, result in the reward function being too difficult to learn with the amount of data and time provided. The exploitation/exploration trade-off was implemented with the help of Gaussian noise. The use of Gaussian noise differs from the original DDPG paper (Lillicrap et al., 2015), because more recent results (Fujimoto, Hoof, and Meger, 2018; Barth-Maron et al., 2018) have shown the original Ornstein-Uhlenbeck noise to be superfluous when compared to its Gaussian alternative. Mathematically, the planner noise can be described as $\mathcal{N}(\mu, \Sigma)$, where μ is a $(17, 1)$ vector of zeros and Σ is the covariance matrix and contains only zeros, aside from on its main diagonal, where it has the corresponding variances(here the variance is 0.7^2 for all diagonal entries). The "planner time noise" is described by the following Gaussian distribution $\mathcal{N}(0, 200)$. Like the other DDPG variants the sparse rewards version is updated each episode using a soft-update where $p = 0.001$.

3.2.1 Reward Structure:

The reward structure for this DDPG version is purposefully simple. We simply pick the planner with the max value from the networks planner vector and check if the time allocated by the network for this planner can solve the current task. If this is the case, we return $R = 10$. If the time allocated is smaller than zero, then we return $R = -10$. Finally, if none of the above is true, the reward would return $R = -1$ (this is not strictly speaking part of a typical sparse reward and is done to disincentivize making too many attempts for a given task). This simple reward structure reduces the probability that we reward sub-optimal intermediary behaviors.

3.3 DDPG Reward Shaping

In our simple reward shaping DDPG implementation, the idea was to motivate beneficial intermediary behaviour, that would hopefully let our agent learn an appropriate policy faster than in our sparse rewards DDPG. The DDPG is used to estimate a planner choice as well as a time allocation for that planner (DDPG estimates a planner and a time). This is different from the DQN variant that only estimated a planner, but the time was fixed. The exploitation/exploration trade-off was implemented with the help of Gaussian noise. A separate noise function was created for the planner noise and the planner time allocation noise. Mathematically, the planner noise can be described as $\mathcal{N}(\mu, \Sigma)$, where μ is a $(17, 1)$ vector of zeros and Σ is the covariance matrix and contains only zeros, aside from on its main diagonal, where it has the corresponding variances (here the variance is 0.7^2 for all diagonal entries). The "planner time noise" is described by the following

Gaussian distribution $\mathcal{N}(0, 200)$. The DDPG is updated each episode using a soft-update where $p = 0.001$ in contrast to the DQN, which uses a hard-update after a fixed number of episodes.

3.3.1 Reward Structure:

The idea behind this reward was to penalize negative times and the overshooting of the time limit. While incentivizing choosing the correct planner or the wrong planner but with time left to solve the task.

The reward for our first DDPG returns -10 iff the current planner time is less or equal to zero, -11 else iff the time missing for any of the planners to be able to solve the task is bigger than the time left, -1 else iff the time left is sufficient for another planner to solve the task, $1 + \frac{timeLeftEpisode}{timePerEpisode - timeForBestPlanner} * 10$ else iff the time missing for the current planner to solve is less than or equal to zero (i.e. solved) and 0 else iff the time missing for the current planner to solve is greater than zero.

The idea behind this reward is to incentivize, assumed to be beneficial, intermediary behaviors such as leaving time for another planner to be able to still solve the task or never estimating times less than zero.

This idea constitutes reward-shaping.

3.4 DDPG Dense Rewards

In our more dense reward shaping version of DDPG the DDPG is used to estimate a vector of planner values as well as a time allocation for each planner. We made this switch so the reward could be based on the entire planner vector and corresponding time vector, as we hypothesized that this would improve the agents rate of learning. The exploitation/exploration trade-off was again implemented with the help of Gaussian noise. Mathematically, the planner noise can be described as $\mathcal{N}(\mu_1, \Sigma_1)$, where μ_1 is a $(17, 1)$ vector of zeros and Σ_1 is the covariance matrix and contains only zeros aside from on its main diagonal, where it has the corresponding variances (here the variance is 0.5^2 for all diagonal entries). The "planner time noise" is described by the following Gaussian distribution $\mathcal{N}(\mu_2, \Sigma_2)$, where μ_2 is a $(17, 1)$ vector of zeros and Σ_2 is a covariance matrix with 200^2 filling every entry in the main diagonal, while every other entry in the matrix is zero. The DDPG is updated each episode using a soft-update where $p = 0.001$.

3.4.1 Reward Structure:

Unlike other reward functions developed for this project, this reward function considers the entire planner vector output by the actor network when determining the reward. The idea behind this was to see if we could make the reward signal more dense by inferring the quality of entries in the planner vector that do not represent the maximum (i.e. the current planner choice by the network). In doing this, we incentivize that the soft-maxed output vector has a more even distribution of values (can be interpreted as probabilities) between the different planners. This should theoretically make the network estimate more realistic values for all

the planners as the tendency to focus on a small group of planners (network tends to fall into a pattern of picking the same planners) is reduced.

For each of the 17 entries in the planner vector, we calculate C_i, K_i and t_{pen_i} and use them to derive R_{a_i} and R_{t_i}. Here, we only clip the time outputs after calculating the planner vector reward which is explained below. Also let x denote the planner values from the network and y the planner times.

- $C_i = 10$ iff planner i can still solve in remaining time else $C_i = -5$
- $R_{a_i} = C_i(2x_i - 1)$
- $K_i = 10$ iff planner i solves with time y_i else $K_i = -10$
- $R_{t_i} = x_i K_i$
- $t_{pen_i} = -3$ iff y_i larger than time left, -5 iff $y_i \leq 0$ else 0

We then calculate the reward for the whole planner vector $R_p = \sum_{i=1}^{n=10}(R_{a_i} + R_{t_i} + t_{pen_i})$. To get our final reward R, we need t_0, which is equal to the time left in the episode iff the current planner can solve the task, iff not, t_0 is equal to the time allocated by the current planner. Then $t = max(0, t_0)$ and t_{prop} is t divided by the remaining time in the episode. Finally $R = R_p t_{prop} - \lambda$, where $\lambda = 1$ is a penalty for taking too many steps and is set to constant value instead of being proportional to the number of steps, because the agent's goal is to maximize the discounted cumulative reward due to the use of the Bellman equation.

3.5 MARL Adhere to Region

Due to the success of our DDPG algorithm for certain toy problems that constituted moving an agent towards a certain region with a strictly continuous action space, we felt that the hybrid nature of the action space could be inhibiting the DDPG agent's learning. To eliminate this potential problem, we use multiple agents (MARL), where a DQN outputs a planner vector (i.e. a planner choice) and a DDPG outputs a time vector (i.e. time allocation for each potential planner choice). Both the DDPG and the DQN have separate reward functions. Both versions of MARL make use of Gaussian noise.

3.5.1 DQN Reward Structure:

- Sort the planners from best to worst for current task based on minimum time still required to solve.
- Cut off planners that don't solve in the remaining time.
- Iff selected planner solves and $max_index_{solved} > 0$ then
  $R = \frac{(max_index_{solved} - index_current_planner)}{max_index_{solved}} 0.5 + 0.5$
  else iff selected planner solves, has index 0 in sorted and $max_index_{solved} = 0$ then $R = 1$
  else doesn't solve $R = 0$.

3.5.2 DDPG Reward Structure:

The DDPG reward works by creating "reward windows", which essentially means defining the boundaries of these windows and their corresponding rewards. The lower bound of the main "reward window" is the best (a.k.a smallest) time for any of the seventeen planners multiplied by 1.1 to give more freedom to the planner choice. Iff the time still required for the current planner is less than the time left in the episode, the upper bound is set to the time for the current planner multiplied by 1.2 to once again give more freedom to the choice of planner by the DQN without completely skewing the window shape. Iff current planner cannot solve the task anymore given the time left in the episode, we reset the upper bound to $upper_bound = lower_bound + 100$, so that the DDPG can still reward good time allocation. We also check that the difference between lower bound and upper bound is at least 50, so that our target window has a somewhat substantial size at least. Iff this is not the case, the upper bound is reset to $upper_bound = lower_bound + 50$. The upper bound is also clipped to be the remaining time in the episode, if it overshoots similiar to how we clip the "current planner time" throughout this project.

- iff planner time is less or equal to 0 $R = -10$
- iff $min_time < planner_time \leq lower_bound$ then $R = \frac{planner_time - min_time}{lower_bound - min_time}$
- iff $lower_bound < planner_time \leq upper_bound$ then $R = 1$
- iff $upper_bound < planner_time < time_left_episode$ then $R = \frac{planner_time - time_left_episode}{upper_bound - time_left_episode}$
- else $R = 0$

3.6 MARL Adhere to MSE

The second version of MARL only differs from the first in its reward function for the DDPG. In this version, we replace "reward windows" with MSE.

3.6.1 DDPG Reward Structure:

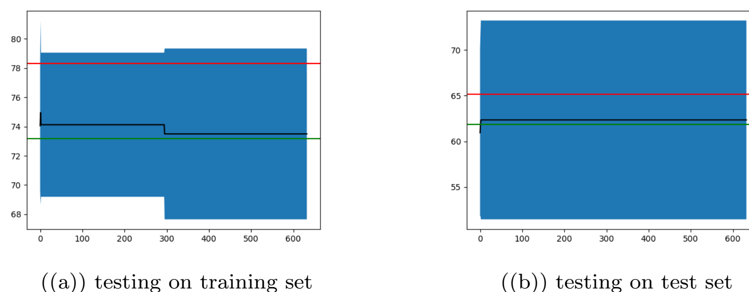
Instead of "reward windows", the reward is based on the mean squared error relative to an expected value. This value is calculated by first sorting an array of planners in ascending order of time remaining. Second, we remove planners that cant solve the task in the remaining time and calculate the median of the resulting array. The reward is then $R = -(planner_time - median)^2$.

4

Experiments

In the below experiments a random action baseline which represents taking random actions was implemented as a comparison. The implementation of this random action baseline is not always the same from agent to agent and is described in more detail in the section of the corresponding experiment. On the y-axis the percentage correct is displayed, while the x-axis shows amount of times that we test the network. The testing of the network is done every ten episodes, meaning the episode number is the x-value multiplied by ten.

4.1 DQN with Constant Action Time



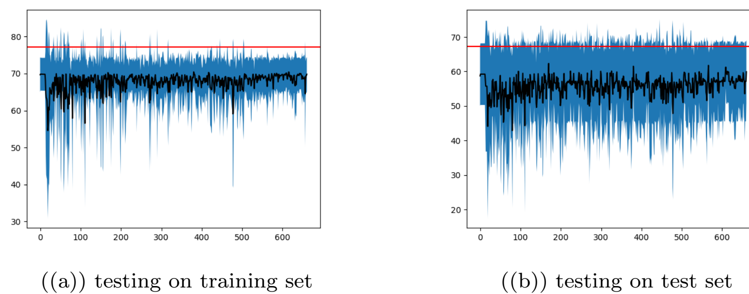
((a)) testing on training set ((b)) testing on test set

Figure 4.1: DQN with constant action time results

Here, the red line represents the random action baseline corresponding to allocating half of the total time to one randomly chosen planner and then trying another randomly chosen planner for the remaining time. The green line represents whole time available being allocated to a single planner that is randomly chosen. Also, our DQN trials tended to converge to local optima relatively quickly. After this point the outputs stagnate, which can be seen by looking at the mean percentage achieved which is represented by the black line. Here we see both for the test set and for the training set that there is little variation over time in the mean, which can be explained by the fact that the individual DQN experiments stagnate relatively quickly. Furthermore, there seems to be a larger standard deviation (represented by the shaded blue region) on the test set than on the training set. This is likely due to

the fact that only ten attempts where made per experiment due to computational limits, because the test set is significantly smaller than the training set, and because the test set varies more in task difficulty (e.g. the sum of all planner times for a given task as a metric for how difficult the task is).

4.2 DDPG Sparse Rewards

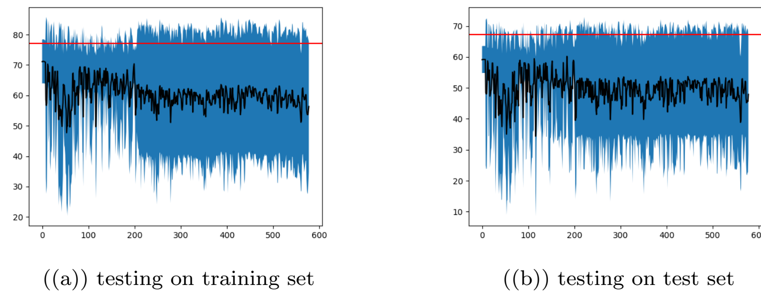


((a)) testing on training set ((b)) testing on test set

Figure 4.2: DDPG (sparse rewards) results

Here red line represents the random action baseline consisting of choosing a random planner for a random amount of time between zero and time remaining repeatedly until the time limit is reached and is the same for all the DDPG and MARL agents. In this version, a slight positive correlation is noticeable between percentage solved and the number of episodes. This would have to be further tested to see if this trend continues for larger episode numbers and to which degree. The standard deviations plotted indicate that the agent would occasionally beat the random action baseline (which is confirmed by looking at the more detailed experiment logs). Also, the standard deviation decreases, as the number of episodes gets larger, which makes sense given the agent has randomly initialized weights at the beginning.

4.3 DDPG Reward Shaping

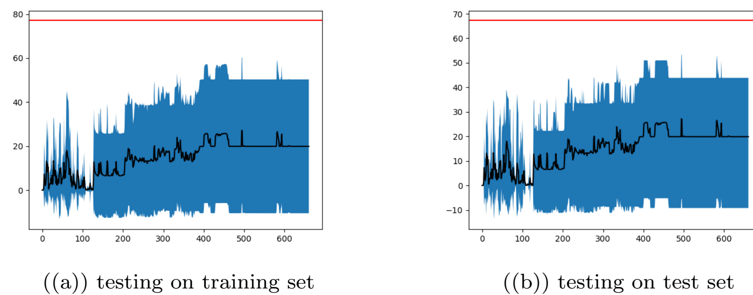


((a)) testing on training set ((b)) testing on test set

Figure 4.3: DDPG (reward shaping) results

By shaping the reward we hoped to encourage beneficial intermediary results that would steer our agent in the right direction. The network is regrettably quite unstable and often gets stuck on strategies far less optimal than the random action baseline. However, rarely, as can be seen by the shaded standard deviation regions, the agent is superior to the random action baseline.

4.4 DDPG Dense Rewards

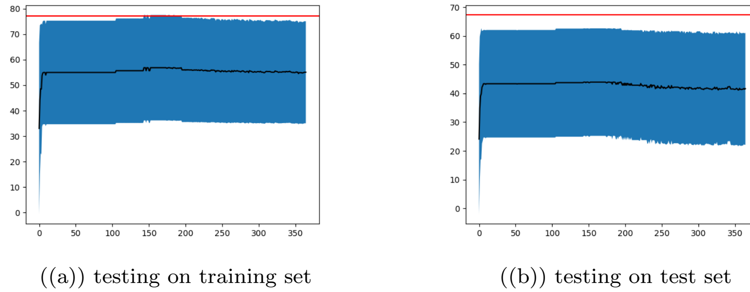


((a)) testing on training set ((b)) testing on test set

Figure 4.4: DDPG(reward whole vector) results

In this version, the idea was to make the reward function much more dense, by rewarding all entries in the planner vector. This led to ”reward hacking” and learning sub-optimal intermediary policies. Even though the mean percentage (represented by the black line) of the experiments increases as the episode number increases, the random baseline is so much higher than the mean percentage that this trend is somewhat irrelevant. In all, this was the least effective/efficient experiment in the time frame tested.

4.5 MARL Adhere to Region

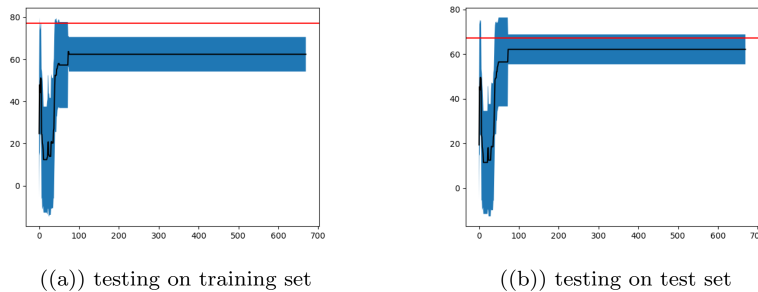


((a)) testing on training set ((b)) testing on test set

Figure 4.5: MARL Adhere to Region results

The red line once again indicates the random action baseline. This multi-agent agent was created with a DDPG to estimate a continuous time output and a DQN to estimate a planner for a task state. The agent fails to make noteworthy progress in the small time frame that it is active, and fails to exceed the random action baseline. Testing on the larger training set yielded better results than on the test set. Here the mean percentage is more stable than for instance the mean percentage for the sparse rewards DDPG.

4.6 MARL Adhere to MSE

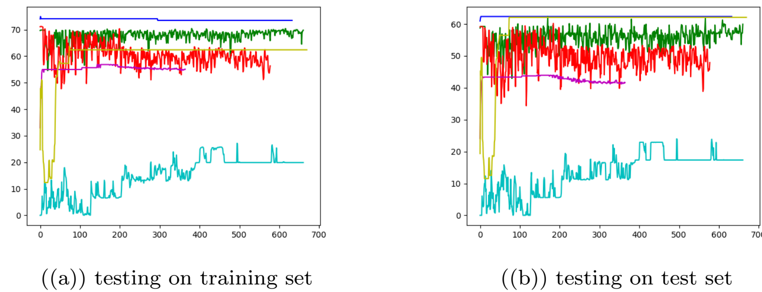


((a)) testing on training set ((b)) testing on test set

Figure 4.6: MARL Adhere to MSE results

The second version of MARL differed only in its reward for the DDPG from the first version. We now replace reward windows with the MSE to an expected value. This leads to a 5%+ average improvement over the reward window version of MARL on the training set and a 15%+ average improvement on the test set. These results are likely due to the more concave nature of the objective function (which makes it easier to optimize over) when using MSE, than when utilizing the reward windows approach.

4.7 Comparison between Experiments



((a)) testing on training set ((b)) testing on test set

Figure 4.7: Comparison of all Agents

In the figure above the blue line represents the DQN, the green line represents the DDPG with sparse rewards, the red line represents the DDPG with reward shaping, the turquoise line represents the dense rewards DDPG, the magenta line represents the MARL agent with "reward windows" and the yellow line represents the MARL agent that uses the MSE.
From the results above we can see that DQN performs the best in terms of percentage correct but the learning rate is stagnant. Sparse rewards DDPG, despite performing worse in the time frame given, seems to exhibit a consistent and slightly positive learning rate, meaning that given a longer time frame it would possibly overtake the DQN. The reward shaping version of DDPG is pretty similiar in its result to the sparse rewards version up until epsiode 2000 (i.e. 200th time testing). Both MARL versions perform somewhat similarly, with the MSE version showing greater deviations, but a notably better final performance. The worst performance is exhibited by the dense rewards version of the DDPG, that upon closer inspection of the reward function allows "reward hacking", which massively hinders the agent's ability to learn the appropriate patterns for its purpose.

5

Summary & Future Work

In this project, we aimed to enhance prior research on online planner portfolios by applying reinforcement learning techniques. Our approach allows multiple attempts to be made per task after only having trained one network. This contrasts with the previously utilized approach, based on supervised learning, by Delfi (Katz et al., 2018; Sievers et al., 2019a), in that it doesn't have to train multiple networks, to be able to make multiple attempts. Some minor changes included adding some additional information to each state that describes the history of attempted planners as well as the time remaining for the current task. We developed various RL agents using various types of algorithms with various reward functions, but unfortunately, the mean accuracy of the agent was exceeded by the random action baseline in all of our experiments. Some potential reasons for this include:

Insufficient Number of Samples

In previous RL successes, such as the Atari DQN (Mnih et al., 2013), the number of samples required to reach this level of proficiency was generally in the millions. In our case, the number of samples was approximately 2000. The planning fallacy states that completing a task will take longer than you think it will. In RL, the planning fallacy is that learning the policy will take more samples than one would expect.

Insufficient Quality of Samples

Previously we stated that a small number of samples can be problematic for learning a policy. In the same way, having low quality samples can lead to a slow or flawed learning rate. In our case, most of the tasks can be solved by most if not all of the planners. If a task can be solved by most planners, then little information can be gained about which planners are "better" based on the current task. In this case, we would say the task has low information density. If an already small task set has low information density, this can complicate the training of an RL agent even further.

Insufficient Stability and Sample Efficiency in Current RL Algorithms

RL agents are famously unstable. The same RL agent will converge on a more than acceptable solution 70% of the time, while devolving to a worse than random policy

the rest of the time. In other areas of machine learning this would not be considered a success from a statistical point of view, but in the current state of RL, this divergence due to the initial conditions is completely normal. Further factors, like a potential lack of consistent feedback for RL algorithms and the tendency for tasks to be more complex, lead to more ways that the model can fail than in traditional supervised learning.

Insufficient Quality of Image Encoding at Representing Patterns in Task

To represent tasks as images, the mathematical object constituting the task is translated to an abstract structure graph, which is then translated to an adjacency matrix which can be encoded as a grey scale image. This translation cannot guarantee information preservation. Using the graphs directly and making use of GNNs circumvents the previously mentioned problem, but does not address the quality of the representation for identifying useful patterns in the data. The previous point can be illustrated by the following example: Assume we have an image I and a bijective function f that randomly maps every pixel in the image to a new pixel (deterministically) to create an image $I^{'}$. Then the information contained in I is a subset of the information contained in the tuple $(I^{'}, f)$. Despite the preservation of information the random jumbling of information in the image space is likely to make it more difficult for a machine to make predictions based on the new image and function than previously. A large part of this is that we are rearranging how certain variables correlate, when jumbling the information. To summarize despite abstract structure graphs encoding the tasks losslessly, we cannot be sure that this representation is well suited to derive patterns from the task using neural networks.

Reward Function Incorrectly Constructed

Constructing or shaping an appropriate reward function for an environment is paramount to the RL agent's success. When shaping a reward, we try to incentivize beneficial intermediary behaviour on the path to a goal state. However, if we mistakenly incentivize the wrong behavior, this will trivially lead to sub-optimal results.

Future work would likely lie in dealing with some of the above mentioned points. Below are a few of the most promising ideas for future work related to the problems mentioned above.

Increase Number of Samples

Increasing the number of samples and compute time for the algorithm is one of the more rudimentary ways in which we can improve performance. The question is how we get these samples. Traditionally, this has been done by using a domain specific task generator, but more recent work has led to different approaches such as AutoScale (Torralba, Seipp, and Sievers, 2021). AutoScale is an automatic tool that selects tasks for a given domain from a set of tasks, that are generated by a traditional task generator. It does this while taking the performance of current planners into account. With an algorithm like AutoScale, it might be possible to not only increase the number of samples, but also increase the quality of samples in the data set.

Prioritized Experience Replay

To deal with the non-uniform distribution of sample quality, a method of prioritizing the sampling of more promising samples over less promising ones was devised (Schaul et al., 2015). The fact that our sample set is relatively small and the that over 50% of our tasks have relatively low quality (in terms of information to be gained), seems to indicate that a prioritized replay memory would be beneficial towards increasing sample usage efficiency.

Progress in RL Algorithms

Over the years since the original DQN Atari paper (Mnih et al., 2013), improvements have been made to algorithms such as DQN and DDPG resulting in algorithms such as double DQN (DDQN) and A3C. Still, for many practical use cases, the stability and efficiency of these algorithms is severely lacking. This makes solving certain tasks using RL infeasible for the moment. To tackle the problem of online portfolios, more progress may be required.

Novel Ways of Translating Automated Planning Tasks to Graphs

As discussed in the previous section, the manner in which we represent planning tasks to the network could have a large effect on the efficiency of the network at identifying usable patterns from the representation. More research into alternative representations of tasks and their properties is likely necessary, to certify that the manner in which we represent tasks is not adding to the accidental complexity of the problem.

Supervised Learning The Delfi project (Katz et al., 2018; Sievers et al., 2019a), which made use of supervised learning, showed more promise at learning desirable patterns from task representations than the RL methods used in this paper. Ideas like training a portfolio of supervised learners or adapting supervised learning in a manner in which it can be possible to make multiple attempts per task, are ideas that have not been tried to the best of our knowledge.

6 Conclusion

The deep RL revolution began with the seminal Atari paper (Mnih et al., 2013) with a multitude of significant improvements in recent years. Inspired by these results, we decided to see if some of these algorithms could be applied to the problem of creating online planner portfolios in automated planning.

We tested a variety of agents and agent types to this effect, but none of the agents exceeded the random action baseline and in fact, all the agents means were exceeded by the random action baseline in terms of percentage correct. The main hypothesized reasons for this unsatisfactory result were outlined in the previous chapter, and lead us to believe that plain vanilla reinforcement learning algorithms are insufficiently stable to be able to deliver viable results for this problem, given the amount of data available at the moment. Even if we assume that the data sets would be large enough, this lack of stability becomes an even bigger problem when using a specific RL algorithm on an action space that it wasn't designed for. DQNs and variants thereof operate well on discrete action spaces, whereas actor-critic networks and policy gradient methods can deal well with continuous action spaces. Despite actor-critic networks doing a decent job of dealing with hybrid action spaces when compared to DQNs, there is still much to be desired. In all likelihood, cooperative MARL approaches will improve our ability of being able to deal with these hybrid action spaces in the coming years.

We also believe that a more informative task set would greatly benefit the agent's learning, because having tasks that almost every possible planner can solve does not tell the agent much about which planners are better, and can lead to a slow and/or sub-optimal learning rate.

In summary, there are many ideas for future work and potential ways to improve the results from this project. The previous work on online portfolios using supervised learning has shown success, which means that supervised learning is perhaps a better candidate for future work in planner portfolios due to the for example far higher sample efficiency and reduced potential for falsely configuring hyper-parameters.

Bibliography

Baker, B.; Kanitscheider, I.; Markov, T. M.; Wu, Y.; Powell, G.; McGrew, B.; and Mordatch, I. 2019. Emergent Tool Use From Multi-Agent Autocurricula. *CoRR*, abs/1909.07528.

Barth-Maron, G.; Hoffman, M. W.; Budden, D.; Dabney, W.; Horgan, D.; Tb, D.; Muldal, A.; Heess, N.; and Lillicrap, T. 2018. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*.

Bellman, R. 1952. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences*, 38(8): 716–719.

Bellman, R. 1957a. *Dynamic Programming*. Dover Publications. ISBN 9780486428093.

Bellman, R. 1957b. A Markovian Decision Process. *Indiana Univ. Math. J.*, 6: 679–684.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced Symmetry Breaking in Cost-Optimal Planning as Forward Search. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 343–347. AAAI Press.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2015. Symmetry Breaking in Deterministic Planning as Forward Search: Orbit Space Search Algorithm. Technical Report IS/IE-2015-03, Technion.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed Model Checking with Distance-Preserving Abstractions. In Valmari, A., ed., *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, 19–34. Springer-Verlag.

Edelkamp, S. 2006. Automated Creation of Pattern Database Search Heuristics. In Edelkamp, S.; and Lomuscio, A., eds., *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, 35–50.

Fujimoto, S.; Hoof, H.; and Meger, D. 2018. Addressing Function Approximation Error in Actor-Critic Methods.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; Wilkins, D.; Barrett, A.; Christianson, D.; et al. 1998. PDDL— The Planning Domain Definition Language. *Technical Report, Tech. Rep.*

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. volume 2, 1007–1012.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. In *ICAPS 2011 Workshop on Planning and Learning*, 28–35.

Irpan, A. 2018. Deep Reinforcement Learning Doesn't Work Yet. https://www.alexirpan.com/2018/02/14/rl-hard.html. Accessed 15/11/2022.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online Planner Selection for Cost-Optimal Planning. In *Ninth International Planning Competition (IPC-9): Planner Abstracts*, 57–64.

Keskar, N. S.; Mudigere, D.; Nocedal, J.; Smelyanskiy, M.; and Tang, P. T. P. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N. M. O.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online Planner Selection with Graph Neural Networks and Adaptive Scheduling. In Conitzer, V.; and Sha, F., eds., *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*, 5077–5084. AAAI Press.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized Experience Replay.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019a. Deep Learning for Cost-Optimal Planning: Task-Dependent Planner Selection. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7715–7723. AAAI Press.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2017. Structural Symmetries of the Lifted Representation of Classical Planning Tasks. In *ICAPS 2017 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, 67–74.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2019b. Theoretical Foundations for Structural Symmetries of Lifted PDDL Tasks. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 446–454. AAAI Press.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient Symbolic Search for Cost-optimal Planning. *Artificial Intelligence*, 242: 52–79.

Torralba, ; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1): 376–384.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8: 279–292.

A

Appendix

A.1 Network Structures

A.1.1 DQN Network Structure:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2,2), stride = (1,1))$.
- $Dropout(dropout_rate = 0.5)$
- $Flatten()$
- $Linear(input_size = 516128, output_size = 100)$
- $Linear(input_size = 135, output_size = 100)$
- $Linear(input_size = 100, output_size = 100)$
- $Linear(input_size = 100, output_size = 17)$

A.1.2 DDPG Sparse Rewards Network Structure

A.1.2.1 Actor:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2,2), stride = (1,1))$.
- $BatchNorm(num_Channels = 32)$
- $Flatten()$
- $Dropout(dropout_rate = 0.49)$
- pre-planner out: $Linear(input_size = 516163, output_size = 100)$
  pre-time out: $Linear(input_size = 516163, output_size = 100)$
- planner out: $Linear(input_size = 100, output_size = 17)$
  time out: $Linear(input_size = 100, output_size = 1)$

A.1.2.2 Critic:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2,2), stride = (1,1))$.
- $BatchNorm(num_Channels = 32)$
- $Flatten()$
- $Dropout(dropout_rate = 0.49)$
- $Linear(input_size = 516163, output_size = 100)$
- $Linear(input_size = 100, output_size = 1)$

A.1.3 DDPG Reward Shaping Network Structure

A.1.3.1 Actor:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2,2), stride = (1,1))$.
- $BatchNorm(num_Channels = 32)$
- $Flatten()$
- $Dropout(dropout_rate = 0.5)$
- pre-planner out: $Linear(input_size = 516163, output_size = 100)$
  pre-time out: $Linear(input_size = 516163, output_size = 100)$
- planner out: $Linear(input_size = 100, output_size = 17)$
  time out: $Linear(input_size = 100, output_size = 1)$

A.1.3.2 Critic:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2,2), stride = (1,1))$.
- $BatchNorm(num_Channels = 32)$
- $Flatten()$
- $Dropout(dropout_rate = 0.5)$
- $Linear(input_size = 516163, output_size = 100)$
- $Linear(input_size = 100, output_size = 1)$

A.1.4 DDPG Dense Rewards Network Structure

A.1.4.1 Actor:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2, 2), stride = (1, 1))$.
- $BatchNorm(num_Channels = 32)$
- $Flatten()$
- $Dropout(dropout_rate = 0.5)$
- pre-planner out: $Linear(input_size = 516163, output_size = 100)$
  pre-time out: $Linear(input_size = 516163, output_size = 100)$
- planner out: $Linear(input_size = 100, output_size = 17)$
  time out: $Linear(input_size = 100, output_size = 17)$

A.1.4.2 Critic:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2, 2), stride = (1, 1))$.
- $BatchNorm(num_Channels = 32)$
- $Flatten()$
- $Dropout(dropout_rate = 0.5)$
- $Linear(input_size = 516197, output_size = 100)$
- $Linear(input_size = 100, output_size = 1)$

A.1.5 MARL Adhere to Region Network Structure:

A.1.5.1 Actor:

- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2, 2), stride = (1, 1))$.
- $Dropout(dropout_rate = 0.50)$
- $Flatten()$
- $Linear(input_size = 516128, output_size = 100)$
- $Linear(input_size = 135, output_size = 100)$
- $Linear(input_size = 100, output_size = 100)$
- $Linear(input_size = 100, output_size = 1)$

A.1.5.2 Critic:

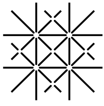
- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2, 2), stride = (1, 1))$.
- $Dropout(dropout_rate = 0.50)$
- $Flatten()$
- $Linear(input_size = 516128, output_size = 100)$
- $Linear(input_size = 137, output_size = 100)$
- $Linear(input_size = 100, output_size = 100)$
- $Linear(input_size = 100, output_size = 1)$

A.1.5.3 DQN:

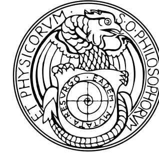
- $Conv2d(number_input_channels = 1, number_output_channels = 32, kernel_size = (2, 2), stride = (1, 1))$.
- $Dropout(dropout_rate = 0.50)$
- $Flatten()$
- $Linear(input_size = 516128, output_size = 100)$
- $Linear(input_size = 135, output_size = 100)$
- $Linear(input_size = 100, output_size = 100)$
- $Linear(input_size = 100, output_size = 17)$

A.1.6 MARL Adhere to MSE Network Structure:

The network structure is identical to MARL adhere to Region.



Faculty of Science



Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: Deep Reinforcement Learning for Online Planner Portfolios

Name Assessor: Dr. Prof. Malte Helmert

Name Student: Tim Goppelsroeder

Matriculation No.: 18-057-711

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: 06.02.2023 Student: Tim Goppelsroeder

Will this work, or parts of it, be published?

- ○ No
- ● Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: 14.02.2023

Place, Date: 06.02.2023 Student: Tim Goppelsroeder

Place, Date: __________ Assessor: __________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis