Saarland University

Bachelor Thesis

Deep Reinforcement Learning with LTL Auxiliary Tasks

Submitted by:

Christian Closheim

Submitted on:

May 30, 2022

Reviewers:

Univ.-Prof. Dr. Verena Wolf

Univ.-Prof. Dr. Holger Hermanns

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

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

Saarbrücken, May 30, 2022

Christian Closheim

Abstract

Deep Reinforcement Learning (DRL) has achieved great success in learning behavioural rules in many different environments. Environments with sparse rewards are particularly difficult because there is little basis for optimizing behaviour. Auxiliary tasks are one approach to address this problem by adding knowledge about environments and improving outcomes.

The difficulty lies in finding appropriate auxiliary tasks that are effective across a variety of different domains. This thesis presents an approach to developing auxiliary tasks based on temporal logic that provide a deeper understanding of the environment.

This thesis investigates the following aspects.

First, it is shown how two auxiliary tasks based on temporal logic were designed and implemented using only two atomic propositions. These auxiliary tasks aim at making the behaviour of an agent safer by enabling it to anticipate risky areas of the environment.

Second, it is shown that a DRL agent extended with these temporal logic encoded auxiliary tasks can outperform the vanilla DRL agent, an DRL agent with an adaptation of the auxiliary tasks from previous related work, and in some cases also a value-based approach DRL agent, in this work represented by the Deep Q Learning (DQN). This improvement is shown in the levels of training robustness, training speed and the resulting agent's performance.

Third, it is shown that the temporal logic encoded auxiliary tasks make the agent's behaviour safer. The agent learns to avoid dangerous situations, reaches the goal more often and moves along safer routes.

Contents

List of abbreviations

A2C Advantage Actor-Critic

A3C Asynchronous Advantage Actor-Critic

CNN Convolutional Neural Networks

CTL Computation Tree Logic

D Deterministic

DQN Deep Q Learning

DRL Deep Reinforcement Learning

DSMC Deep Statistical Model Checking

DZ Danger Zone

FP Feature Prediction

LSTM Long Short Term Memory

LTL Linear Temporal Logic

LTS Labeled Transition Systems

MDP Markov Decision Processes

N Noisy

NS Normal Start

PC Pixel Control

RL Reinforcement Learing

RP Reward Prediction

RS Random Start

UCZ Unavoidable Crash Zone

UNREAL UNsupervised REinforcement and Auxiliary Learning

VR Value Replay

1. Introduction

Deep Reinforcement Learning (DRL) has achieved amazing successes in learning near-optimal policies for sequential decision-making problems in recent years. It has been applied to various applications such as Go and Chess [35, 36, 37] or Atari games [29, 30]. DRL uses an action-feedback loop of self-playing and observing the environment. The current strategy is improved by reinforcing decisions that lead to good performance. A reward signal from the environment is used to define performance, which can be either positive, negative or neutral. Decisions that lead to positive rewards are reinforced, and those that lead to negative rewards are mitigated. The problem is that in a variety of environments, a positive reward signal is rarely (or never) observed during the exploration phase in sparse reward settings [2, 19, 25, 32, 33].

Auxiliary tasks are not directly related to the agent's behaviour, but help build a more expressive representation of the environment state. Their positive effects on learning efficiency and quality of the final policy have been previously studied in the context of DRL [22, 24, 39].

Appropriate auxiliary tasks may improve the dynamics and stability of representational learning, which has been shown to be useful in sparse reward situations [22, 26]. The challenge is to design effective auxiliary tasks.

The work of Jaderberg et al. [22] addressed this issue by introducing the UNsupervised REinforcement and Auxiliary Learning (UNREAL) agent. The UNREAL agent learns auxiliary tasks that provide a more meaningful representation of the environment. The corresponding neural network representations are more meaningful because the policy for reward maximization and the auxiliary tasks share weights.

Some tasks used by Jaderberg et al. [22] can not be easily transferred to any other benchmarks, as they rely on video input, such as from the Atari games. For example, the pixel control auxiliary task predicts the pixel changes in the video input and learns to maximize it. This is based on the assumption that large pixel differences are indicators of large rewards. This task cannot easily be applied to important applications such as autonomous driving relying on sensor input. In general, it still remains unclear what constitutes a good auxiliary task and how one can design appropriate tasks that can improve the learning procedure and the resulting agent's overall performance.

To enable the design of general tasks that are not limited to features relevant to a specific application, like video input or specific sensor data, this thesis presents a method that uses Computation Tree Logic (CTL) to define auxiliary tasks in the context of DRL. CTL provides a framework that builds on *atomic propositions* instead and allows to systematically design tasks that allow inferences about the environment. These *atomic*

propositions are better generalizable than other environmental features.

Two auxiliary tasks are presented that can be easily applied to many sparse reward situations: Learning when an agent unavoidably violates a safety condition and learning when there is a possibility of violating a safety condition. This work shows that the CTL encoded auxiliary tasks perform better than a version of the Advantage Actor-Critic (A2C) [38] and better than the adaptation of the UNREAL presented in this work. The same quality features that Jaderberg et al. [22] have investigated are examined: robustness of training, speedup of learning progress, and performance. Furthermore, it is shown in this work that the CTL encoded auxiliary tasks make the behaviour of the agents safer.

This work is structured as follows. In chapter 2, some basics about Reinforcement Learing (RL) and temporal logic are explained. Chapter 3 summarizes the results of Jaderberg et al. [22] and presents their approach. Chapter 4 introduces the environment used for training and evaluation. Chapter 5 defines the used CTL encoded auxiliary tasks. The experimental setups are explained in chapter 6 and the results are presented in chapter 7.

2. Basics

In the first part of this chapter, the basics of Reinforcement Learing (RL) and Deep Reinforcement Learning (DRL) are presented. The intuition behind RL, possibilities of assessing behaviour as well as first algorithmic approaches are explained.
In the second part of this chapter, the basics of temporal logic are explained.

2.1. Reinforcement Learning

This section begins with a brief introduction to the basic elements of RL and then presents a policy-based agent.

2.1.1. Introduction to Reinforcement Learning

The point of RL is an agent learning to control its behaviour in an environment in such a way that it maximizes a reward signal. The agent can perform various actions, observes how the environment changes as a result, and receives a reward every time it acts. The agent has no predefined knowledge of which actions to perform. Instead, it has to discover through trial and error which actions result in the greatest overall reward. Regularly, different actions not only influence the immediate reward, but also determine which subsequent states the agent will visit and therefore the possible future rewards. This influence on subsequent rewards is called delayed rewards. Trial-and-error search and delayed reward are the most important aspects of RL [38].

A Markov Decision Processes (MDP) is given by a tuple $(S, A, R, s_0, p, \gamma)$ and let $t \in \{0, 1, \dots\}$. An agent observes a sequence of *states* $s_t \in S$ and *rewards* r_{t+1} as a result of choosing *actions* $a_t \in A(s_t)$, where s_0 is the initial state, $s_{t+1} \sim p(\cdot \mid s_t, a_t)$ is the next state and $r_{t+1} = R(s_t, a_t, s_{t+1})$ is the reward obtained in state s_t when choosing action a_t. Note that the s_t, a_t and r_t are considered to be random variables [4].

A discount factor γ is used to take into account the time interval between different rewards. This discount factor $\gamma \in (0; 1]$ is exponentiated with the steps that have passed to reach this reward. The overall return

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

for a time step t, is the accumulated discounted reward for all future states [38].

A key aspect of RL is the policy, which defines the behaviour of the agent. The policy is a mapping from states to actions. In the human psyche, one would speak of association [38]. The policy is the core element of any RL agent, as it alone determines the agent's behaviour both in training and in evaluation [38]. Policies can be deterministic or stochastic. A stochastic policy $\pi(a \mid s)$ is a state-dependent probability distribution over actions that probabilistically chooses one of the available actions in a state, while a deterministic policy always chooses the same action for the same state.

The policy, therefore, influences which action the agent prefers in which state and thus implicitly which rewards the agent will earn in the future. The action value

$$Q^{\pi}(s;a) = \mathbb{E}[G_t|s_t = s; a_t = a]$$

is the expected future return if the agent selects action a in state s and follows policy π [28].

Therefore, for each state, one can also define a value function

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) Q^{\pi}(s;a).$$

This value function weighs the possible action values according to their probability by the policy [38].

With this value function, it is possible to compare different policies with each other. A policy π' behaves better or equal than a policy π if

$$V^{\pi'}(s) \geq V^{\pi}(s) \forall s \in S.$$

The policy π', therefore, achieves an equal or higher return from each state [38].

If there is a ranking between different policies, then there is at least one policy which performs better than or equal to every other policy in the space of all possible policies. In the literature, this optimal policy is usually referred to as π^*. It does not necessarily have to be a single one, because several policies might be able to reach a maximum reward. The aim is to learn the (approximately) optimal policy [38].

Modern approaches use deep artificial neural networks for function approximation. These approaches are called DRL [38]. The detailed explanation of deep learning would go beyond the scope of this work.

2.1.2. Advantage Actor-Critic

Deep learning policy-based methods [38] directly model a parametrized policy $\pi(a_t|s_t;\theta_\pi)$ as an *actor* network, where θ_π are the weights of the neural network. This *actor* network

takes states as input and returns a probability distribution over actions as output. Advantage Actor-Critic (A2C) [38, 28] uses a separate *critic* network to approximate a value function $v(s_t; \theta_v)$. θ_v and θ_π are not necessarily disjunct, as it is possible to use two separate neural networks or some shared weights [28].

The optimization of the neural networks is done by applying a stochastic gradient ascent to their parameters θ_π and θ_v [11]. Given sufficiently accurate estimates, the gradients are of the form $\nabla_{\theta\pi} v_{\pi_\theta \pi}(s_t)$ for the *actor* and $\nabla_{\theta v} A(s_t, a_t; \theta_\pi, \theta_v)^2$ for the *critic* network, where $A(s_t, a_t; \theta_\pi, \theta_v)$ is defined below.

While the policy is used to execute actions in the environment and to gain experience, the value function is used as a baseline for optimizing the policy. For this the advantage function

$$A(s_t, a_t; \theta_\pi, \theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$$

is used, which expresses an advantage of a particular action over the mean value of that state [38]. The A2C uses an n-step return, meaning that an update of the neural network occurs after at least n environment steps, or when an episode is finished. For terminal states, the part $\gamma^k V(s_{t+k}; \theta_v)$ in the advantage function becomes zero, because there is no future return. For non-terminal states, on the other hand, this part estimates the future return to benefit from the previous experience before the termination of an episode [22].

For the optimization, a loss function must be defined, which is to be minimized or maximized [11]. These loss functions express the deviation of the estimation by the approximated function to an expected value. In A2C these loss functions are $\mathcal{L}_\pi = \mathbb{E}_{(s_t,a_t)\sim\pi} [\log \pi(a_t|s_t; \theta_\pi) A(s_t, a_t; \theta_\pi, \theta_v)]$ for the *actor* and $\mathcal{L}_v = \mathbb{E}_{(s_t,a_t)\sim\pi} [A(s_t, a_t; \theta_\pi, \theta_v)^2]$ for the *critic* [22]. This uses the property that the advantage function is also a measure of the error of the value function prediction [28].

The gradients are therefore calculated as

$$\nabla_{\theta\pi} \mathcal{L}_\pi = \mathbb{E}_{(s_t,a_t)\sim\pi} [\nabla_{\theta\pi} \log \pi(a_t|s_t; \theta_\pi) A(s_t, a_t; \theta_\pi, \theta_v)]$$

for the *actor* and

$$\nabla_{\theta v} \mathcal{L}_v = \mathbb{E}_{(s_t,a_t)\sim\pi} \left[\nabla_{\theta v} A(s_t, a_t; \theta_\pi, \theta_v)^2\right]$$

for the *critic* [22].

Asynchronous Advantage Actor-Critic

The version Asynchronous Advantage Actor-Critic (A3C) [28] uses multiple threads, working in parallel on several versions of the environment. For this, training is done with copies of the neural network parameters. With each update of the global parameters,

these are then copied into the local parameters. This way, every thread benefits from the accumulated experience of all threads.

Algorithm 1 A3C pseudocode [28].

Let θ_π and θ_V be global shared parameter vectors
Let θ'_π and θ'_V be thread-specific parameter vectors
Initialize step counter $t \leftarrow 1$
Initialize global shared episode counter $E \leftarrow 1$
repeat
  Reset gradients: $d\theta_\pi \leftarrow 0$ and $d\theta_v \leftarrow 0$
  Synchronize thread-specific parameters $\theta'_\pi = \theta_\pi$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta'_\pi)$
    Receive reward r_t and mew state s_{t+1}
    $t \leftarrow t + 1$
    this
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ'_π: $d\theta_\pi \leftarrow d\theta_\pi + \nabla'_{\theta_\pi} \log \pi(a_t|s_t;\theta'_\pi) A(s_t, a_t; \theta'_\pi, \theta'_v)$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (A(s_t, a_t; \theta'_\pi, \theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ_π using $d\theta_\pi$ and of θ_v using $d\theta_v$
  $E \leftarrow E + 1$
  this
until $E > E_{max}$ =0

Algorithm 1 illustrates a pseudocode of the A3C for one working thread. One can see how the rewards found are used to calculate the optimization of the policy and the value function using the gradients. There are multiple ways of an asynchronous update of the global parameters, but explaining them would go beyond the scope of this work.

Entropy Regularization

When exploring the environment, an agent must always balance between exploration and exploitation [38]. Exploitation uses the knowledge found to reproduce strategies, whereas exploration is necessary to find new states that may have even better outcomes. If the agent only exploits, the result might be suboptimal, if the agent only explores, it does not use the previous knowledge and might make the same mistakes more often,

wasting training time. Therefore, it is important to balance between the two extremes.

In deterministic algorithms such as Q-learning [38], the agent always follows the action that leads to the state with the highest value. Since this would lead to pure exploitation, an additional exploration policy is usually used which simply selects a random action in a uniformly distributed manner. The agent changes over time from this exploration policy to a deterministic q-value policy [38].

With probabilistic algorithms like A2C or A3C, no second policy needs to be used to balance exploration and exploitation. The choice of action is based on a probabilistic distribution. This has the advantage of not having to switch between pure determinism and pure randomness, and can integrate a certain amount of knowledge into the probability distribution of individual actions. To ensure exploration, the probabilistic policy must not become deterministic, meaning that the total probability must not focus on a single action [38].

During training, the policy converges through optimization and focuses on actions with a high expected reward more likely. In extreme cases, the policy converges to a nearly deterministic policy. To ensure that the policy does not converge too rapidly to a possibly suboptimal policy, it has been proven useful to add the entropy of the policy to the loss [28].

Entropy is a measurement of uncertainty [34]. For a discrete set of actions A where $p(a_1), \ldots, p(a_n)$ are the probabilities for the actions $a_1, \ldots, a_n$ the entropy is defined as:

$$H(A) = -\sum_{a \in A} p(a) \log p(a).$$

This formula is at its maximum if all probabilities are equally distributed [34]. This is the maximum of uncertainty because there is no information that any action has an advantage over another.

In Figure 2.1 a minimal example of an action space with two possible actions is displayed. As can be seen, entropy is highest where both actions have the same probability and decrease on both sides until it reaches zero if the choice becomes deterministic. In this case, there is no uncertainty about which action is taken.

For the policy-based agents, the entropy $H(\pi(s_t; \theta_\pi))$ expresses the uncertainty about the correct action in a state s_t. If one integrates the entropy into the *actor* loss, one obtains

$$\mathcal{L}_\pi - \mathbb{E}_{s_t \sim \pi}[\alpha H(\pi(s_t; \theta_\pi))]$$

where α is a weighting factor [22]. This regularization of the *actor* prevents a too rapid convergence to a possible suboptimal solution [22].

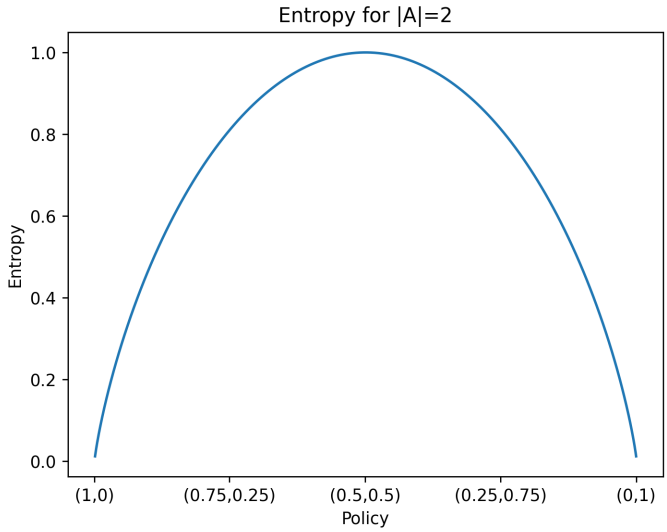


Figure 2.1.: Example of how the entropy changes for different policies in an actions space with two possible actions.

Usually the policy and the value function do not use two separate parameter sets θ_π and θ_v but a common parameter set θ. Therefore, the entire A2C and A3C loss is of the form:

$$\mathcal{L}_{A3C} \approx \mathcal{L}_v + \mathcal{L}_\pi - \mathbb{E}_{s_t \sim \pi}[\alpha H(\pi(s_t; \theta))]. \tag{2.1}$$

This common loss is the basis for extensions, by auxiliary tasks.

2.2. Temporal Logics

To formalize temporal characteristics, atomic propositions are used. For this the MDP is extended to $(S, A, R, AP, L, s_0, p, \gamma)$ where the function $L : S \to 2^{AP}$ labels states with atomic propositions [4]. Intuitively, atomic propositions are the representation of simple known facts about the states of the system under consideration [4]. These facts can be either true or false.

Temporal logics are defined on Labeled Transition Systems (LTS) [4], so an LTS can be constructed from an MDP with policy π, where a transition from state s to state s' exists, iff $\exists a \in A : p(s'|s, a) \cdot \pi(a|s) > 0$, which means that there exists an action a for which the probability to reach s' is greater 0.

Linear Temporal Logic (LTL) [4] is a logic to describe conditions on temporal paths with atomic propositions. This makes statements on these paths possible. The definition

of LTL grammar in Backus Naur form over a set of atomic propositions AP is

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc \phi \mid \phi_1 U \phi_2$$

where $a \in AP$ [4].

The connectives $\bigcirc$ and U are called temporal modalities. $\bigcirc\phi$ holds if ϕ holds in the next state. $\phi_1 U \phi_2$ holds if there is some state in the future for which ϕ_2 holds and ϕ_1 holds at all moments until that future state [4]. There are other temporal modalities and binary operators that can be derived from this basic grammar, but this would go beyond the scope of this work.

Computation Tree Logic (CTL) is an extension of LTL. While LTL describes a single possible future, CTL extends this and takes different futures into account in a tree-like structure [27]. The definition of CTL state formulae grammar in Backus Naur form over a set of atomic propositions AP is

$$\Phi ::= \top \mid a \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists\phi \mid \forall\phi$$

where $a \in AP$ and ϕ is a path formula formed according to the grammar

$$\phi ::= \bigcirc\Phi \mid \Phi_1 U \Phi_2$$

where Φ, Φ_1 and Φ_2 are state formulae [4].

The connectives $\exists$ and $\forall$ are called path quantifiers. $\exists\phi$ holds if there exists a path for which ϕ holds. $\forall\phi$ holds if ϕ holds on all possible paths from this state. The path quantifier and temporal modalities need to combine to build a legal CTL state formula. For example, $\exists \bigcirc g$ holds if there exists a path on which g holds in the next state.

3. Unsupervised Reinforcement and Auxiliary Learning

This chapter explains an extension of the A3C [28]. Jaderberg et al. [22] published the UNsupervised REinforcement and Auxiliary Learning (UNREAL) agent in 2017. In the following, their approach, as well as algorithmic details, are explained and the experimental results of Jaderberg et al. [22] are presented.

3.1. Approach

The UNREAL agent is based on the A3C [28] algorithm. The approach is to use contextual information to improve the learning results as well as the robustness. This use of context information is called an auxiliary task, as it is not directly used to improve the policy, but to build a more meaningful representation of the environment that goes beyond the mere policy or state value [22].

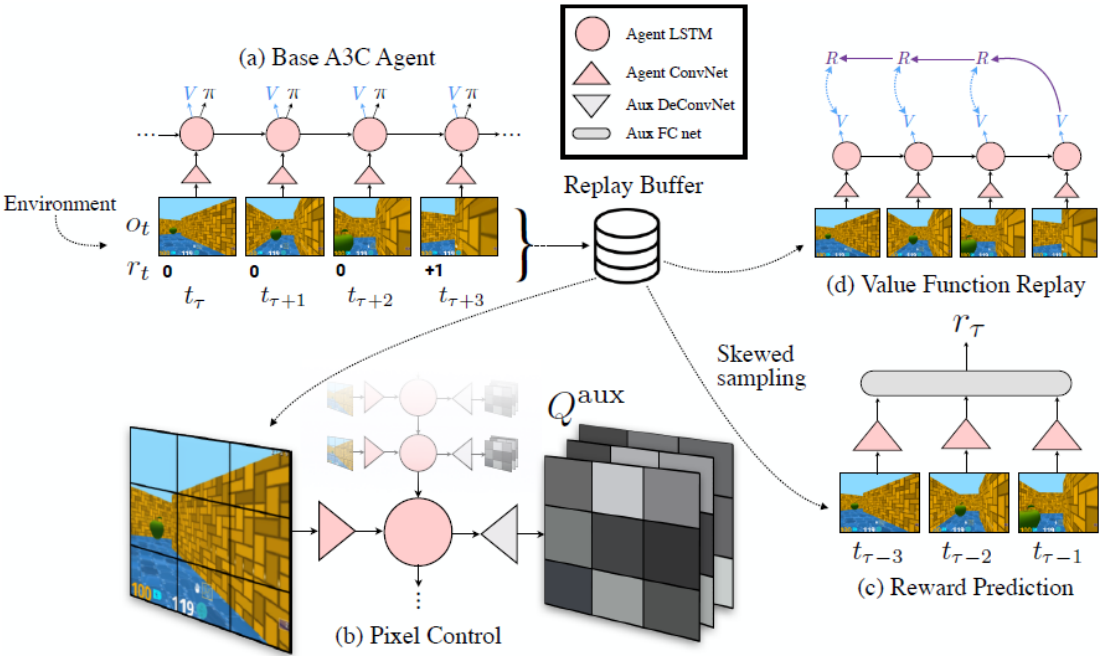


Figure 3.1.: Illustration of the UNREAL agent structure from Jaderberg et al. [22].

An illustration of the UNREAL agent can be seen in Figure 3.1. In the top left corner, the basic A3C task is shown. It works as introduced in section 2.1.2. In this case, the

goal was to learn how to move in a 3D environment only using the image input of the environment [22].

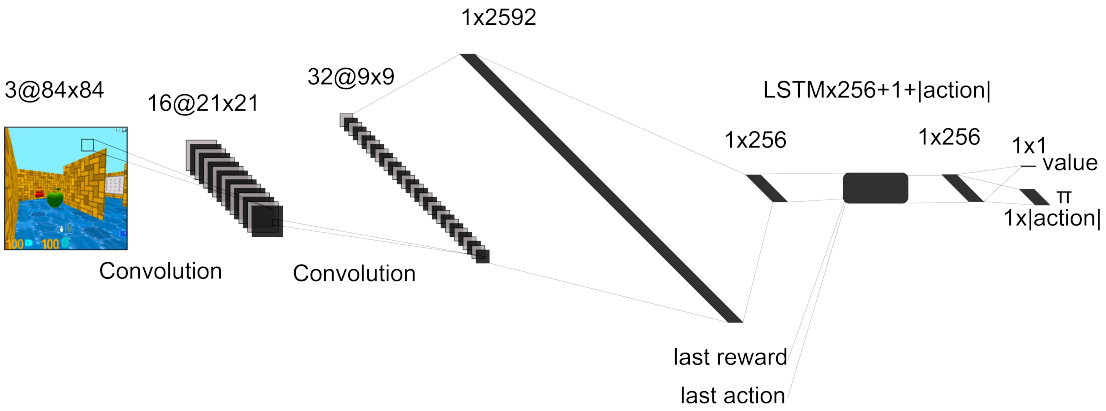


Figure 3.2.: The structure of the A3C neural network based on description from Jaderberg et al. [22, 1].

The neural network for the basic A3C task, shown in Figure 3.2, uses convolutional layers with filters to map the RGB coloured image input to feature layers [22].

Convolutional layers are a special type of neuronal layers, as they work in two dimensions. This is necessary, because in images not only are two neighbouring pixels in one line highly connected in information gain, but also neighbouring pixels in columns. Therefore, these convolutional layers work on small windows that want to identify some features, like edges, in the image, and on deeper levels more complex hyper-features [31].

The output of the convolutional layers is mapped through a linear layer to 256 neurons. These 256 neurons combined with the last reward and last action are the input for the Long Short Term Memory (LSTM) [22].

A LSTM is a special structure in neural networks that allows the neural network to build memory about temporal dependencies [10, 21]. The explanation of LSTM would go beyond the scope of this work.

The output of the LSTM is mapped by linear layers to the value and policy output [22].

3.2. Auxiliary Tasks

The UNREAL agent uses three additional off-policy auxiliary tasks. The auxiliary tasks do not require any extra information other than that used by the A3C. The images and rewards of the environment, as well as the selected action, are stored in a replay buffer. This stored data is used for auxiliary tasks [22].

The auxiliary tasks are the Pixel Control (PC) task, the Reward Prediction (RP) task and the Value Replay (VR) task. To accomplish the optimization, the different tasks share some neural network structures. This leads to each task optimizing the neural network for its purpose, which in total leads to a faster and better improvement [22]. In the following, each task is explained in detail.

3.2.1. Pixel Control Task

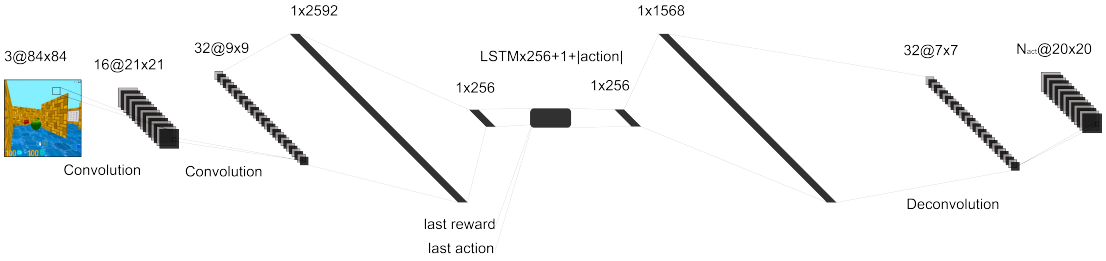


Figure 3.3.: The structure of the PC neural network based on description from Jaderberg et al. [22, 1].

The PC task is trained to predict changes in pixels and therefore to maximize the changes because reward critical events often occur with special visual changes. The structure of the network is visualized in Figure 3.3. It uses the same convolutional and LSTM layer as the A3C network but processes the output of the LSTM to images using deconvolutional layers. This is a special technique to produce images, which processes an image for every possible action. As shown, no complete image is generated in full resolution, instead, the focus lies on the centre of the image. By this, the network learns which action maximizes the difference between the pixels of the image, whereby the agent learns to influence the environment [22].

Jaderberg et al. [22] experiment with an additional feature control task, which uses the activation of a hidden feature layer of the neural network itself as an auxiliary reward, which enables the agent to behave in such a way that certain neurons are activated. However, this was not part of the final UNREAL agent version [22].

3.2.2. Reward Prediction Task

The RP task is trained to predict the expected reward in the next step, using three consecutive images as input. The structure of the network is visualized in Figure 3.4. It uses the same convolutional layers as the A3C for all three images, combines their output, and uses one linear layer to map the convolutional output to three neurons using a softmax function to produce a probability output. This network predicts the probability of the next reward being positive, negative or zero [22].

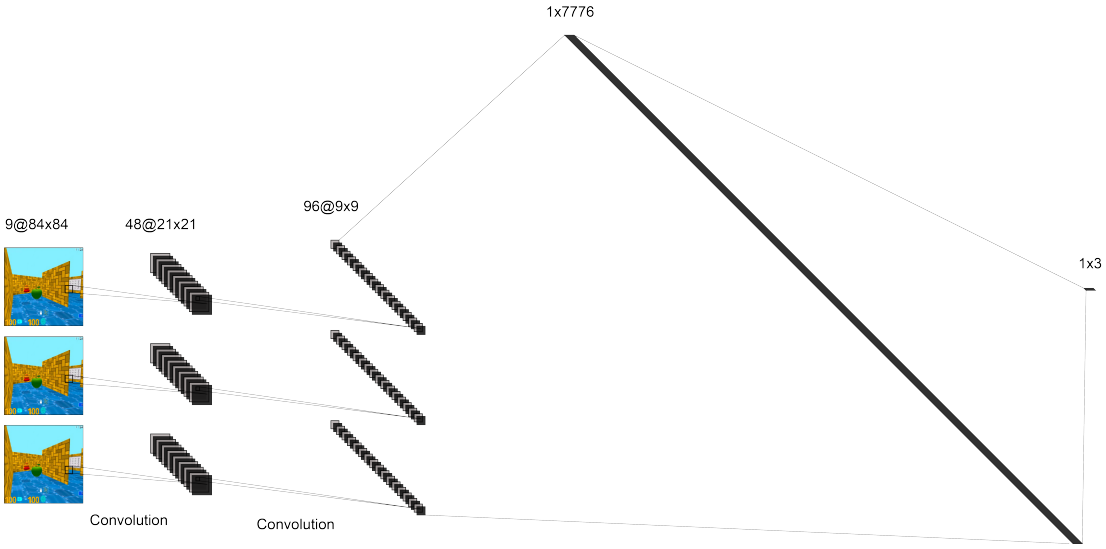


Figure 3.4.: The structure of the RP neural network based on description from Jaderberg et al. [22, 1].

Since the rewards are not evenly distributed in most environments and zero rewards are much more frequent than negative or positive rewards, it is impossible to simply sample randomly from the replay buffer because this would over-represent the zero rewards. Therefore, skewed sampling is used for this task, which handles the problem of sparse rewards by returning non-zero rewards in half of the cases. This means that zero and non-zero rewards are equally frequent in the sample [22].

3.2.3. Value Replay Task

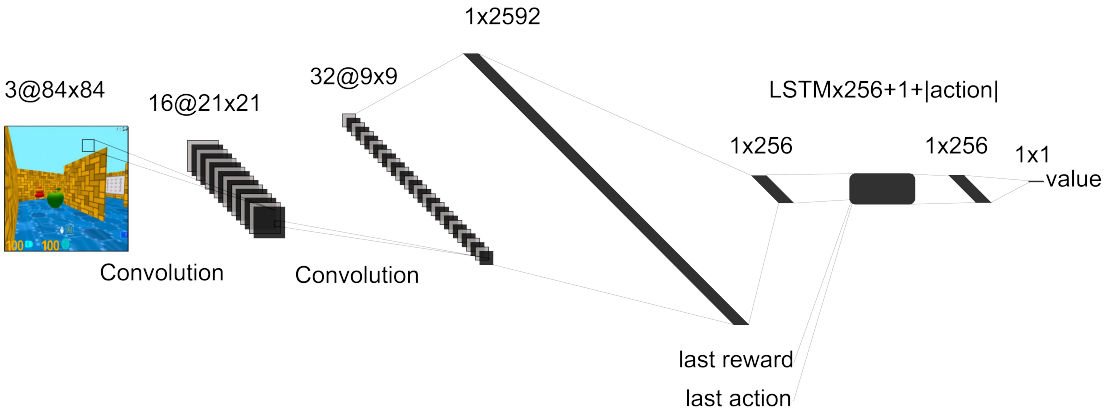


Figure 3.5.: The structure of the VR neural network based on description from Jaderberg et al. [22, 1].

The VR task executes value function replay, which optimizes the value function again with data from the replay buffer. Since the value function is used as a baseline for optimizing the policy, this task is important because the better the approximation of the value function, the better the optimization of the policy becomes. The structure of the network is visualized in Figure 3.5. It uses the same network architecture as the A3C, but without output for a policy. This task uses the replay buffer in a non-skewed manner and variates the temporal position of the inputs to break the dependencies between the inputs, which is unavoidable when optimizing the value function in A3C [22].

3.2.4. Combining

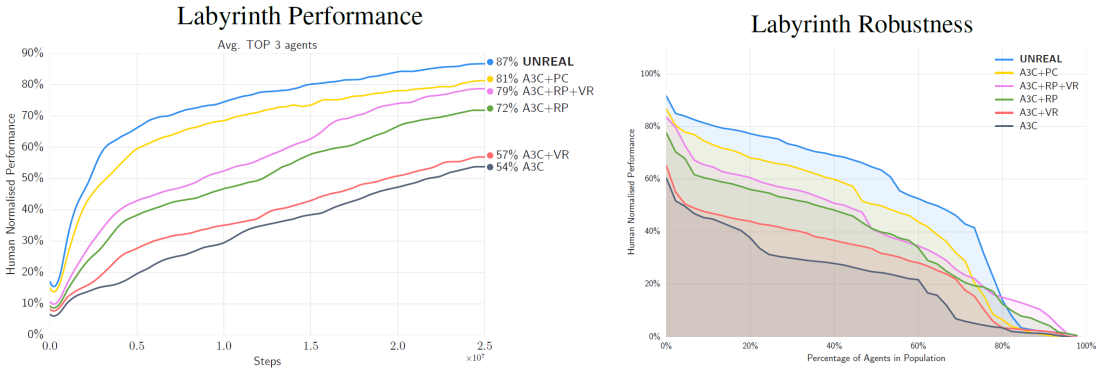
The A3C combined with all three auxiliary tasks results in the UNREAL agent. For the optimization, the auxiliary tasks are performed every time the A3C is updated, which means for an n-step return every n environment step. Therefore, a combined loss is computed, which includes the A3C loss, as introduced in equation 2.1, and the loss of each auxiliary task. This combined loss

$$\mathcal{L}_{UNREAL}(\theta) = \mathcal{L}_{A3C} + \lambda_{VR}\mathcal{L}_{VR} + \lambda_{PC}\sum_{c}\mathcal{L}_{Q}^{c} + \lambda_{RP}\mathcal{L}_{RP}$$

where $\lambda_{VR}, \lambda_{PC}$ and λ_{RP} are weighting factors for the auxiliary task, is then used to optimize the total UNREAL neural network [22].

3.3. Experimental Results

The UNREAL agent was applied to the 3D environment *Labyrinth* and also to 57 *Atari* games [22]. The authors also applied some variants by recombining the set of auxiliary tasks.



(a) Performance (b) Robustness

Figure 3.6.: The results of performance and robustness of UNREAL in *Labyrinth* [22].

The UNREAL agent used for the experiments uses a 20-step return and synchronizes every 20 steps, over 32 parallel threads. Accordingly, the auxiliary tasks are performed every 20 steps, too. The replay buffer stores the last 2000 steps. For the experiments with the environment *Labyrinth*, the weights for the auxiliary tasks are set to 1 for λ_{VR} and λ_{RP} and the weight λ_{PC} is sampled from a log-uniform distribution between the values 0.01 and 0.1. For *Atari* games, a distribution between 0.0001 and 0.01 is used. The learning rate is sampled from a log-uniform distribution between 0.0001 and 0.005 and the entropy costs are also sampled from the log-uniform distribution between 0.0005 and 0.01 [22].

The results are shown in Figure 3.6. Figure 3.6a shows how each auxiliary task, and also each combination, causes a performance improvement over the basic A3C. Figure 3.6b also illustrates how each auxiliary task leads to better robustness. Robustness here denotes the stability of the UNREAL agent against a variation of the learning rate and the entropy weighting. The best single increase in performance was applied by the PC auxiliary task. The UNREAL agent, which includes all three auxiliary tasks, applied the best overall improvement. In the *Atari* games, the results are equal, but the superiority of UNREAL over the combination of A3C with RP and VR task is no longer as prominent [22].

These results are slightly put into perspective by more recent findings. Silver et al. [40] reimplement the auxiliary tasks and test them on other benchmarks against their approach with a general value function. It seems that especially the PC and RP tasks are dependent on the environment and do not lead to a general improvement. These auxiliary tasks performed no better than a random general value function [40]. These results show that these auxiliary tasks do not improve on every benchmark and are therefore task-specific.

4. Racetrack

Instead of *Labyrinth* or *Atari*, this work uses the *Racetrack* benchmark [3, 5, 6, 13, 18] to train and evaluate the agents. *Racetrack* is a grid world game based on a pen and paper game. The grid world contains only a small sample of different fields, which can be combined into an infinite number of different maps of different sizes and complexity. These are goal fields, start fields, free fields and walls. Usually, but not necessarily, several starts and goal fields are placed next to each other to form a start line and a goal line respectively. Multiple goal lines are also possible.

The agent has two goals. Firstly, to reach the goal line without crashing into a wall, and secondly, to do this with as few steps as possible. Learning the first goal is supported by the distribution of the reward. The learning of the second goal is achieved through a discount factor. This factor reduces the positive reward the longer it takes the agent to reach the goal, but also the negative reward, so that the agent learns to survive longer [18].

In the following, interaction with the environment, some game rules, as well as different game modes, are explained. The version used is identical to the version used by Gros et al. [12, 15], therefore the descriptions have been largely adopted.

4.1. Interaction

There are two elements of interaction. One is the action that the agent sends to the environment, the other is the feedback that the agent receives from the environment. Both are explained below.

4.1.1. Actions

The agent does not always perform one step into a neighbouring field as usual in other grid world games, rather it has a velocity like a car, which can be influenced by actions. Since *Racetrack* is a grid world, the velocity $v = (v_x, v_y)$ is a integer pair of x and y dimension speed. The agent can speed up, slow down or maintain velocity in both dimensions by a maximum of one unit in each step [12].

All possible combinations result in nine different actions, as shown in Figure 4.1. The numbers in the picture are the codes that the agent sends to the environment to interact with it [12].

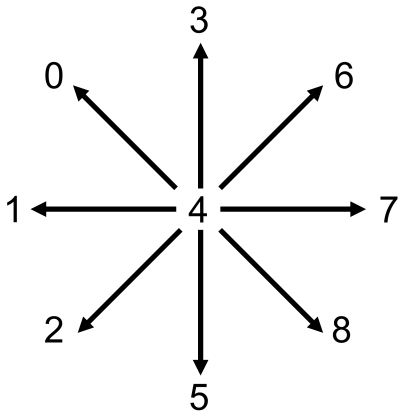


Figure 4.1.: All nine possible actions [12].

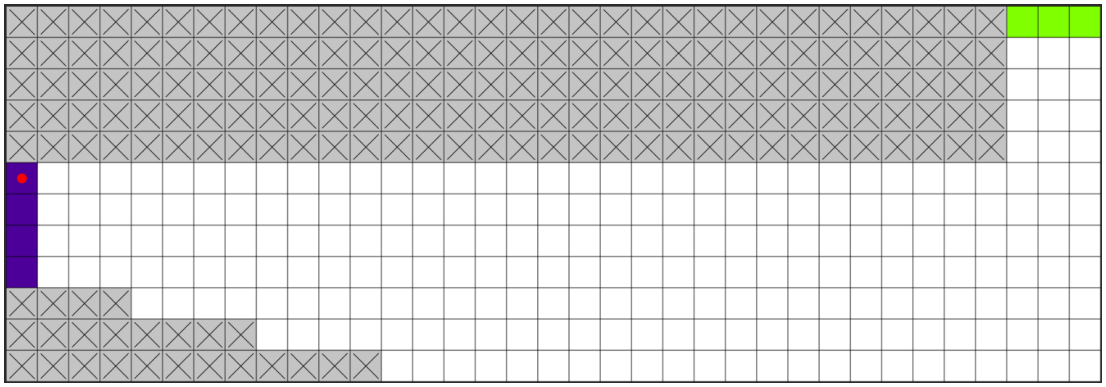
4.1.2. Feedback

The feedback contains two elements. One is the current state of the game that the agent has reached after the last action. The other is the reward the agent received for the last action.

State

The current state of the game depends on the map, the current position and the current velocity of the agent. Combined, this information makes the state of the game fully observable, so that no temporal context from previous states is needed to understand the current state [12]. In other games, such as *Labyrinth*, the state of the game is not fully observable because the agent never sees the whole map [22].

Racetrack provides two kinds of state representation. The first is a representation of the current state as a map. In Figure 4.2a we see the visual representation for human players. This map is called Barto-small [12]. The image representation distinguishes between the green goal line, the purple start line and the white free fields. The walls are marked with an x. However, as this is a lot of unnecessary information, there is another image representation that can be used as an input for especially CNN-based agents. This can be seen in Figure 4.2b. This pixel map only distinguishes between a wall (light grey), the current position (dark grey) and the goal line (white). The free fields are black. Since it is not possible to deduce the velocity from the static image, there is also a velocity vector given on top of the pixel map [12].



(a) Coloured version visualized for human users.

(b) Greyscale version to use in Convolutional Neural Networks (CNN).

Figure 4.2.: *Racetrack* visual output example with a map named Barto-small [18].

The other kind of representation is a feature vector. The feature input,

$$(\underbrace{P_x, P_y}_{\text{Position}}, \underbrace{V_x, V_y}_{\text{Velocity}}, \underbrace{WD_0, WD_1, WD_2, WD_3, WD_5, WD_6, WD_7, WD_8}_{\text{Wall Distance}}, \underbrace{GD_x, GD_y, GD_m}_{\text{Goal Distance}})$$

contains 15 different variables describing the current state. The vector is composed of information about the position, the velocity and further information about the wall and goal distances. Experiments have shown that it is possible to predict the individual features with a CNN from the pixel map [18].

Reward

The reward structure on *Racetrack* is adaptable. To achieve comparable results, the reward structure that has been found to be the best on the Barto-small map through previous studies is used [12].

The *Racetrack* is a sparsely rewarded environment. The agent only receives a non-zero reward at the end of an episode. The atomic propositions *goal* and $safe$ are used for the reward structure. The first one expresses that the agent has reached the goal and the second one that the agent is still alive. The reward structure

$$R\left(s \xrightarrow{(a_x.a_y)} s'\right) = \begin{cases} 100 & \text{if } s' = goal \\ -20 & \text{if } s' = \neg safe \\ 0 & \text{otherwise} \end{cases}$$

expresses that the agent receives a negative reward if it crashes into a wall or leaves the map and a positive reward if it hits or crosses the goal line [18]. For all other steps, it receives a reward of 0.

4.2. Game Modes

The *Racetrack* provides different game modes. It distinguishes between a Normal Start (NS) and a Random Start (RS) mode. In the NS mode the agent is placed randomly on the start line, whereas in the RS mode it is placed on an arbitrary free field on the map. Furthermore, there is a Deterministic (D) and Noisy (N) mode. In the D mode, nothing unforeseen happens. The environment is working as explained before. In the N mode, it can happen with a predefined probability that the applied action fails, and the velocity does not change. This could be especially dangerous in states close to a wall, where the agents need to brake sharply to avoid a crash. This results in four different combinations of modes: RS D, RS N, NS D and NS N [12].

4.3. Rules

The game starts by placing the agent on the start field or randomly on the map, depending on the game mode. Then the agent receives the state representation of the start state. Now the agent performs several steps.

Each step starts by choosing an acceleration encoded in the action number. Then the acceleration $a = (a_x, a_y)$ is added to the current velocity

$$v'_x = v_x + a_x, \text{ and } v'_y = v_y + a_y.$$

With this new velocity $v' = (v'_x, v'_y)$ the agent moves in a straight line to the new target field $p' = (x', y')$ with

$$x' = x + v'_x \text{ and } y' = y + v'_y.$$

Diagonal lines are calculated using the sign function. All visiting points $(x, y), \ldots, (x', y')$ on the way from the start to the goal field are in the track T with

$$T = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}), (x_n, y_n) \rangle$$

such that,

$$T = \begin{cases} \langle (x, y) \rangle & \text{if } v'_x = 0 \text{ and } v'_y = 0 \\ \langle (x, y), (x + \sigma_x, y), (x + 2 \cdot \sigma_x, y), \ldots, (x', y') \rangle & \text{if } v'_x \neq 0 \text{ and } v'_y = 0 \\ \langle (x, y), (x, y + \sigma_y), (x, y + 2 \cdot \sigma_y), \ldots, (x', y') \rangle & \text{if } v'_x = 0 \text{ and } v'_y \neq 0 \\ \langle (x, y), (x + \sigma_x, \lfloor y + m_y \rceil), (x + 2 \cdot \sigma_x, \lfloor y + 2 \cdot m_y \rceil), \ldots, (x', y') \rangle & \text{if } v'_x \neq 0 \text{ and } v'_y \neq 0 \\ & \text{and } |v'_x| \geq |v'_y| \\ \langle (x, y), (\lfloor x + m_x \rceil, y + \sigma_y), (\lfloor x + 2 \cdot m_x \rceil, y + 2 \cdot \sigma_y), \ldots, (x', y') \rangle & \text{if } v'_x \neq 0 \text{ and } v'_y \neq 0 \\ & \text{and } |v'_x| < |v'_y| \end{cases}$$

where $\sigma_x = sgn(v'_x), \sigma_y = sgn(v'_y)$ and $m_x = \frac{v'_x}{|v'_y|}, m_y = \frac{v'_y}{|v'_x|}$ [12].

To reach the goal, the agent must not hit a wall or leave the playing field. If it does, it loses. Each field along the line on which the agent moves counts. This means that if the line crosses a wall, the agent hits it. The agent wins if it hits the goal line on its way. If an agent crashes, it cannot win anymore and vice versa. The first event that occurs counts [12].

5. LTL and CTL Auxiliary Tasks

It is still a challenge to design appropriate tasks because it remains unclear what makes a good auxiliary task. On one hand, the tasks should not be application-specific, but generally applicable. On the other hand, they should not be so abstract that there is no longer any control over the behaviour to be learned.

To allow for the design of general tasks that are not limited to features relevant to a particular application, in this chapter CTL encoded auxiliary tasks for DRL benchmarks with sparse reward structures are presented. Due to the nature of the tasks described here, LTL was not sufficient, therefore it had to be extended to CTL. The minimum requirement for these tasks is that the environment can be described by at least two atomic propositions: *safe* and *goal*. This set of atomic propositions is not specific for *Racetrack* and can therefore easily be applied to other benchmarks. The atomic proposition *safe* describes that the agent is still alive and *goal* describes that the agent has reached the goal. These atomic propositions are used to label the underlying transition system of the MDP.

In this chapter, two CTL auxiliary tasks are introduced. These are called Danger Zone (DZ) and Unavoidable Crash Zone (UCZ). This chapter will start with an intuition part, which describes the motivational background of these tasks. Then the tasks are formally defined and special corner cases are explained. In the end, it is explained how these CTL auxiliary tasks are algorithmically implemented.

5.1. Intuition

The N condition of the *Racetrack* is like driving on ice. Sometimes the agent can not break or speed up, by random chance. This can be dangerous, especially in border conditions when the agent has to brake sharply to avoid crashing into a wall. And like a car, these circumstances need some adaption in the driving behaviour, just like one can not drive in snow and ice the same way one can on a dry road. The danger comes from inappropriate speed. This means that the choice of velocity must be made depending on the situation. In large open spaces, one can accelerate faster with less risk than on winding roads.

The UCZ describes a state in which the agent can no longer avoid a crash, regardless of which policy it would use. Predicting this state is interesting because once the agent is in it, there is no way out and it is bound to lose the game. If the agent can predict this zone, it may also be able to avoid these states.

The DZ on the other hand, describes how many motor failures can happen before the agent drives into a wall. This means that the higher the degree of the DZ, the safer the agent. Therefore, the DZ is determined by the current position as well as the current velocity. If the velocity is higher, the agent takes more fields in one step. Therefore, fields far away from the wall can be dangerous as well. This leads to the fact that the DZ is highly dynamic. For all possible velocity vectors, the DZ map is different. This high dynamic and the fact that there is no easy way to calculate the DZ directly from the states make it an interesting metric.

5.2. Definition

In this section, the UCZ and DZ are formally defined and the relationship between these two zones is explained.

5.2.1. Unavoidable Crash Zone

The agent is considered to be in the UCZ when it currently is still safe, but it is unavoidable to reach a state where the proposition *safe* does not hold any further. No matter what actions the agent decides to take, all paths lead to a state where the agent is unsafe. To be able to better differentiate the UCZ, a distinction is made between different degrees of the UCZ, i.e., the number of steps the agent can take at a maximum before it unavoidably becomes unsafe. The first UCZ is therefore given by

$$\mathrm{UCZ}_1 = \forall \bigcirc \neg \mathit{safe}.$$

Analogously, the other unavoidable crash zones are defined. For example, $\mathrm{UCZ}_3 = \forall \bigcirc \bigcirc \bigcirc \neg \mathit{safe}$.

5.2.2. Danger Zone

More interesting than unavoidably reaching an unsafe state is the DZ, where a crash is possible, but may still be avoided. The DZ also distinguishes between different degrees. It is defined as

$$\mathrm{DZ}_1 = \exists \bigcirc \neg \mathit{safe},$$

i.e., an agent is in danger zone degree 1, if there is a chance that it crashes within one step. Analogously, $\mathrm{DZ}_3 = \exists \bigcirc \bigcirc \bigcirc \neg \mathit{safe}$.

5.2.3. LTL Simplification for *Racetrack*

Since in *Racetrack* it is mainly interesting when a motor failure becomes dangerous and not an arbitrary action, it is possible to reduce the DZ to an LTL description. So an LTS is used where only the failure path transitions are considered. All other paths are removed before. Therefore, LTL can be used. The first DZ can therefore be redefined to

$$\mathrm{DZ}_1 = \bigcirc \neg \mathit{safe}$$

and analogously, $\mathrm{DZ}_3 = \bigcirc \bigcirc \bigcirc \neg \mathit{safe}$.

5.2.4. Implications

The two zones are related together. It holds that

$$\mathrm{UCZ}_1 \subset \mathrm{DZ}_1 \tag{5.1}$$

because in DZ_1 the agent crashes if a motor failure happens, but there are some areas where it can avoid crashing by braking. In the UCZ_1, it also crashes if a motor failure happens, but there is no way to avoid this. Therefore, $\mathrm{UCZ}_1 \implies \mathrm{DZ}_1$.

The LTL simplified DZ follows the rule

$$\mathrm{DZ}_n \implies \bigcirc \mathrm{DZ}_{n-1} \tag{5.2}$$

because only the motor failures transit between the DZ. Of course, the agent can jump over some DZ by applying other actions. But this is not considered in the LTL simplified definition because this is under the control of the agent and not determined by the probability of a motor failure.

The UCZ follows the rule

$$\mathrm{UCZ}_n \implies \forall \bigcirc (\neg \mathit{safe} \vee \bigvee_{i=1}^{n-1} \mathrm{UCZ}_i). \tag{5.3}$$

This means that from a UCZ_n, the agent can only reach a UCZ of a smaller degree or crash. Therefore while equation 5.1 holds there is no generalization for it.

The UCZ is not a general subclass of DZ so it holds that

$$\mathrm{UCZ}_n \not\subset \mathrm{DZ}_n \tag{5.4}$$

because the degree of the UCZ depends on the best path and the degree of DZ on one path, which is not necessarily the best.

5.3. Corner Cases in *Racetrack*

The LTL simplified DZ entails two corner cases. First, if the agent is standing still. For velocity $V = (0, 0)$ the agent will never drive into a wall by motor failures, therefore the DZ is undefined. Second, if the agent drives through the goal with the motor failures instead of into a wall the DZ is undefined, too.

The same applies to the UCZ. If a path with which a safe state with $V = (0, 0)$ can be reached exits, the agent can stand still there for all time. This means that a state is not an UCZ as long as the goal can be reached from it or a state with $V = (0, 0)$ can be reached.

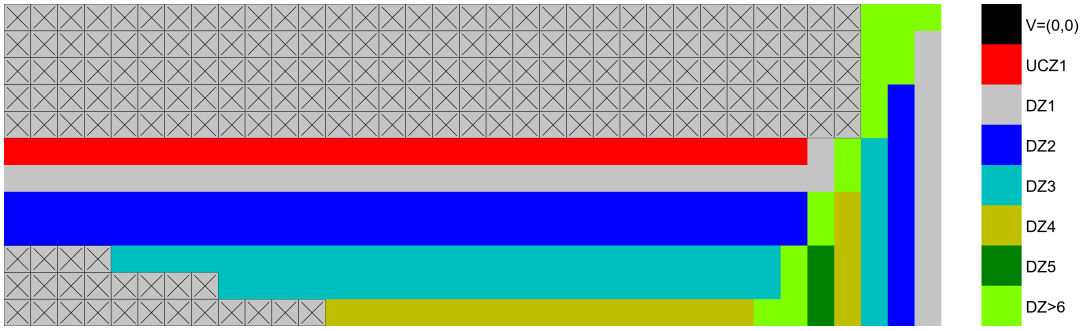
The definition gap in UCZ is not a problem, since we want to avoid this condition and only consider if the agent is in a specific degree of UCZ. But the main idea of the DZ is that the agent rides on the ridge of the DZ degree because this is the safest path for many motor failures. But if the agent follows the highest point of the DZ, the two corner cases are indistinguishable from each other, yet of a considerably different character. Standing still is an undesirable condition that should be avoided while rolling to the goal should be preferred. Therefore, the definition gap will redefine by assigning the degree -1 to states with $V = (0, 0)$ and ∞ to goal-reaching states by motor failures.

5.4. Illustration

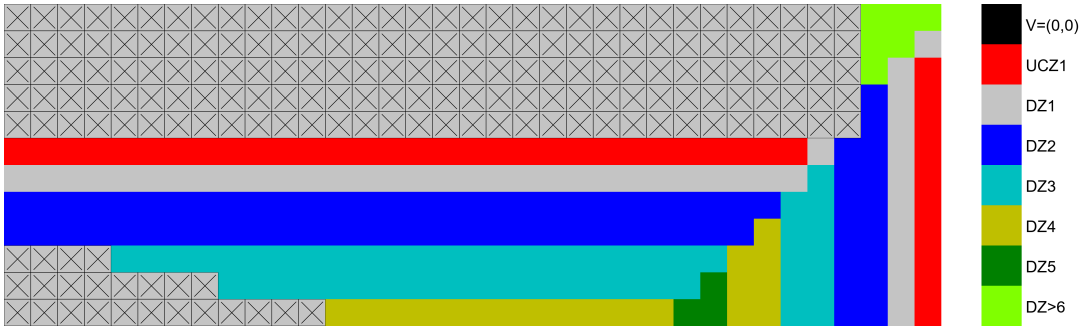
In Figure 5.1a and Figure 5.1b the influence of velocity on the DZ and UCZ distribution is illustrated. The illustrations also depict how the DZ and UCZ would change by speeding up to the right. The illustrations also show the implications of the equation 5.2 and equation 5.4.

In Figure 5.1c it is illustrated how many DZ the agent passes on its way. This is an optimal path in the NS D mode. Especially in the last state before the goal line, the agent is at a high risk of motor failure.

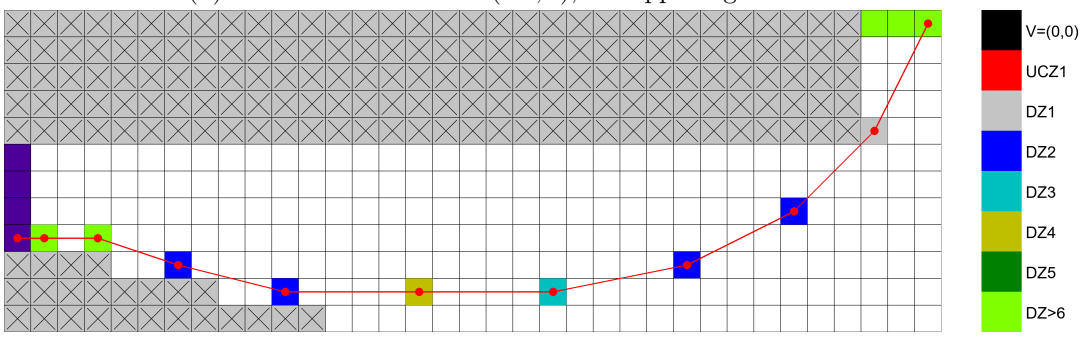
Figure 5.2 displays some minimal examples for the UCZ and illustrates the implications of the equation 5.3. The agent starts on the purple start line with a velocity $V = (1, 6)$. The light green line shows the best path by continuous breaking. This illustrates that the agent can not avoid the crash after a maximum of 5 steps. The red line illustrates that the agent can directly transit from a UCZ_5 to a UCZ_1 by applying a suboptimal action. The blue line illustrates that the agent can directly crash from UCZ_4 by a suboptimal action.



(a) DZ and UCZ for $V = (-2, 1)$, i.e. upper right acceleration.



(b) DZ and UCZ for $V = (-2, 2)$, i.e. upper right acceleration.



(c) DZ for an optimal path in the D mode.

Figure 5.1.: Examples of DZ and UCZ for different velocities and path dependence.

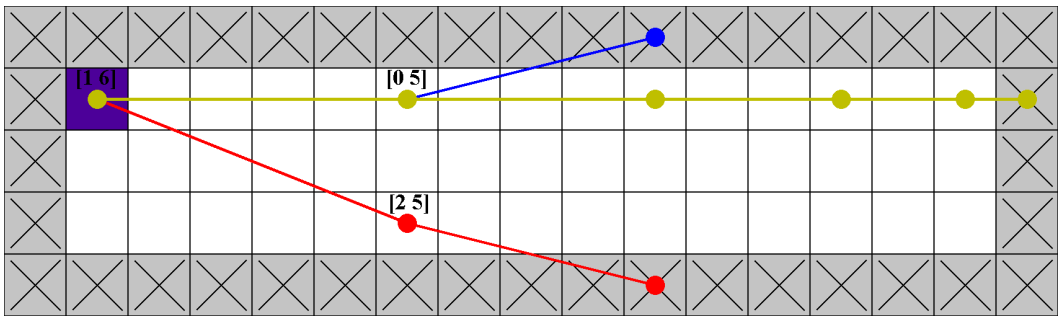


Figure 5.2.: Example how the agent can transit form UCZ_5 to UCZ_1 (red) or crash from UCZ_4 (blue).

5.5. Algorithms

Since the degree of the UCZ can only be calculated by an exhaustive tree search, a simplification of some criteria is used. It is necessary to consider all possible paths to prove a UCZ, but a single path that either leads to a $V = (0, 0)$ or *goal* is sufficient to prove that it is not a UCZ.

Algorithm 2 UCZ calculation pseudocode.

```
 0: function UCZ(state, max_depth)
 1: if state.goal ∨ ¬state.safe then
 2:   return undef
 3: else
 4:   Create queue Q # States considered in the current iteration
 5:   Create queue W # Child states stored for the next iteration
 6:   Q.append(state)
 7:   C = 0
 8:   while ¬Q.empty() ∧ C <= max_depth do
 9:     state = Q.pop()
10:     for each child in state.childs() do
11:       if child.V = (0, 0) ∨ child.goal then
12:         return undef
13:       end if
14:       if child.safe then
15:         W.append(child)
16:       end if
17:     end for
18:     C = C + 1
19:     Q = W
20:     W.clear()
21:   end while
22:   if Q.empty() then
23:     return C
24:   else
25:     return undef
26:   end if
27: end if
```

Therefore, the Algorithm 2 tries to falsify the UCZ. If this is not possible, it is a UCZ. This makes the depth search much more efficient. For large depth, this search is expensive, since the search tree has a fanout of nine in the *Racetrack*. Therefore the algorithm provides to search for different degrees of UCZ so that it is possible to limit the depth of the search. In this case, it is not a UCZ if the agent is still *safe* in one of the leaves.

Algorithm 3 DZ calculation pseudocode.

```
0: function DZ(state)
1: if state.goal ∨ ¬state.safe then
2:   return undef
3: else
4:   if state.V = (0, 0) then
5:     return −1
6:   else
7:     C = 0
8:     while ¬state.goal ∧ state.safe do
9:       state = state.action(4) # see Figure 4.1
10:      C = C + 1
11:    end while
12:    if state.goal then
13:      return ∞
14:    else
15:      return C
16:    end if
17:  end if
18: end if
```

For the LTL simplified DZ, an exhaustive search can be used because it is a linear search tree. It is therefore only necessary to follow this linear tree to a terminal state. The special case of $V = (0, 0)$ is not important, because a motor failure will not change anything. Because of this, the Algorithm 3 will first check if $V \neq (0, 0)$ before the search starts.

6. Experimental Setup

Using the methods presented in the previous chapters, the UNREAL has been adapted so that it can act on the *Racetrack*. These modifications are presented in the first part of this chapter. After that, another auxiliary task was created which predicts the DZ and UCZ. The integration of this task is explained in the second part. The last part explains how the different agents were trained on *Racetrack*.

6.1. UNREAL on Racetrack

To handle differences between the environments, some adaptions to the UNREAL agent were applied. The focus of this work is on the *Racetrack* with feature array representation of the current state. Therefore, this work does not consider convolutional neural networks, but only simple feed-forward networks.

6.1.1. A2C as a Basic Algorithm

As the basic algorithm, an A2C is used instead of an A3C. There were two reasons for this decision. Firstly, it is quite debatable where the A3C draws its advantage from. The general assumption is that the different threads break the correlations between the observed states and thus stabilize learning [28]. Other findings lead to the assumption that A3C decreases the training time, but does not necessarily improve the final results [9, 23], and since the training time is not to be taken into account, the A2C is used. Secondly, the results should be exactly reproducible, which is not possible in A3C. The reason for this is that the A3C is influenced by the scheduler of the CPU and thus there is no control over the order in which the threads are processed.

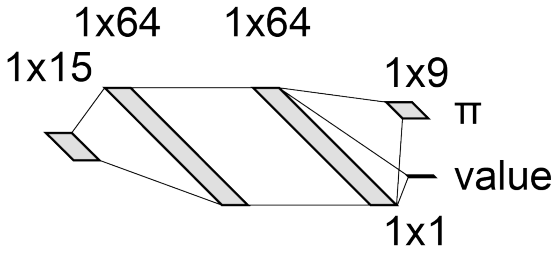


Figure 6.1.: Basic A2C network adaption for *Racetrack* [1]

To obtain comparable results, the neural network's structure is based on that of the Deep Q Learning (DQN) agent [12], which is used as a reference. This has a very simple structure with 2 hidden layers. Only the output layers differ. Just like UNREAL, this version uses a common neural network for the policy and value function approximation. On *Racetrack*, there is no need for LSTM because as explained in section 4.1.2, the state of the game is fully observable and therefore does not need to memorize past states. The resulting network is depicted in Figure 6.1. The calculation of the loss does not change and is done exactly as described in equation 2.1.

6.1.2. Feature Prediction instead of Pixel Control

The PC task is not applicable on the *Racetrack*. The *Racetrack* provides numerical values instead of graphical input. Therefore, instead of a PC task, a Feature Prediction (FP) task was created, which predicts all nine feature vectors of the nine possible successor states for an input feature.

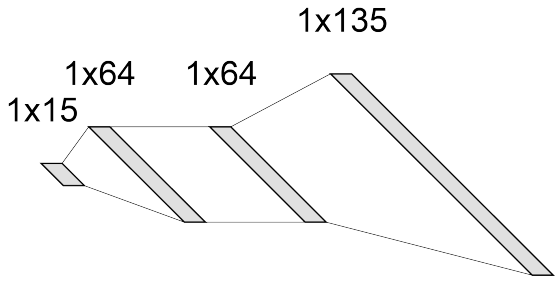


Figure 6.2.: Basic FP network adaption for *Racetrack* [1].

The neural network FP: $\mathbb{R}^{15} \rightarrow \mathbb{R}^{135}$, for predicting action-dependent next-state features, shares the same input and hidden layers as the basic neural network. Only the output layer now consists of 135 output neurons, used to predict $|A| = 9$ times the 15 features. Since for the adaptation of the UNREAL, just like in the template, no further information than the observed ones should be used, consequently not the entire output can be used in the loss function but only the observed next state. Therefore, the loss function for the FP task is a mean squared error loss with the form

$$\mathcal{L}_{FP} = \mathbb{E}_{(s,a,s')\sim\mathcal{U}(B)}\left[\|FP(s,a;\theta) - s'\|_2^2\right]$$

where (s, a, s') are previously experienced transitions sampled uniformly from the replay buffer, $\mathcal{U}(B)$ is the uniform distribution over the replay buffer B and $FP(s, a; \theta)$ the parametrized prediction of the next state given the current state s and an action a. The FP task is performed on a random batch from the replay buffer every time the auxiliary tasks are performed.

6.1.3. Reward Prediction

The RP works in the same way as introduced in section 3.2.2. Instead of the CNN output the output of the second hidden layer is used for the reward prediction. The task is still using three consecutive state observations as input, since Jaederberg et al. [22] also implemented it this way in their version of the UNREAL agent [22].

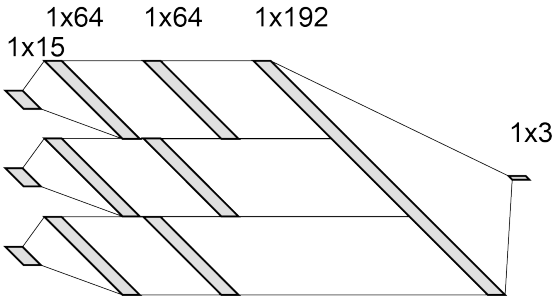


Figure 6.3.: Basic RP network adaption for *Racetrack* [1].

The neural network RP: $3 \cdot \mathbb{R}^{15} \rightarrow [0,1]^3$, is shown in Figure 6.3. The neural network shares all input and hidden layers with the basic network. Only the third hidden layer, which links the output of the three hidden layers, is task-specific. Just like in the original UNREAL [22], no exact reward is predicted here. Instead, a rewarding class RC is predicted, whether the next reward is positive, negative or zero, which on *Racetrack* is equivalent to winning, losing and still being alive.

The loss function is multiclass cross-entropy of the form

$$\mathcal{L}_{RP} = \mathbb{E}_{(s,s',s'',r)\sim\mathcal{S}(B)} \left[- \sum_{rc \in RC} (-|0|+)_r \log RP(rc|s, s', s''; \theta) \right]$$

where $RP(rc|s, s', s''; \theta)$ is the probability of the reward class under three observations and $(-|0|+)_r$ as a binary indicator $(0|1)$ if the real observed reward r is negative, zero or positive. This is summed up over all three predictions. But since the reward classes are distinct, because a reward cannot be in two classes at the same time, a large part of the calculation is omitted by the zero factors. What remains is only the negative logarithmic probability of the observed reward class.

As in the original UNREAL [22], a skewed sampling $\mathcal{S}(B)$ of the replay buffer is used, which returns zero and non-zero rewards equally often. However, problems can arise when a non-zero reward only occurs at the end of each episode. Therefore, it can happen that there are not enough samples in the replay buffer. In this case, no RP optimization is performed. This problem can occur if the replay buffer is too small, the batch size is too large or the episodes are too long.

6.1.4. Value Replay

The VR works in the same way as introduced in section 3.2.3.

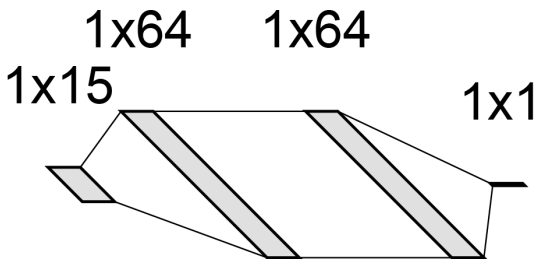


Figure 6.4.: Basic VR network adaption for *Racetrack* [1].

The task uses the same neural network structure VR: $\mathbb{R}^{15} \rightarrow \mathbb{R}$ as the basic A2C algorithm without the policy output layer, as illustrated in Figure 6.4. The loss function is a mean squared error loss with the form

$$\mathcal{L}_{VR} = \mathbb{E}_{(s_t, s_{t+k}, r_t, \ldots, r_{t+k-1}) \sim \mathcal{U}(B)} \left[\left(\left(\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta') \right) - V(s_t; \theta) \right)^2 \right]$$

where $V(s_{t+k}; \theta')$ is set to zero if s_{t+k} is terminal and k is the maximum step size minus the current count t. θ' are the parameters used when making this experience and θ are the current parameters. The first part estimates the return for the state under the current policy (if the episode terminates) and the second part is the current prediction by the function approximation. In the implementation, the first part is calculated after an episode is terminated for each state in the replay buffer. Therefore, there is no need to remember old parameters. The optimization is applied to a random batch from the replay buffer, not on sequences of 20 steps as in the original version by Jaderberg et al. [22].

6.2. Danger Zone and Unavoidable Crash Zone Prediction

As described in chapter 5, the DZ is a metric for how vulnerable to crashing due to a motor failure the agent is in the current state. The DZ_UCZ task aims to predict the next DZ based on the feature input. The task predicts the different degrees of the DZ and additionally the UCZ_1 as an auxiliary task with shared weights.

However, no information is lost through this joint output. By the implication of equation 5.1, one can conclude that if the agent is in a UCZ_1 then it is also in a DZ_1. In this task, the basic observations are not sufficient to optimize the predictions. The

CTL and LTL functions are checked by investigating the necessary transitions by using the simulation of the *Racetrack*.

Since no more than a single value is to be used, the DZ and UCZ must be bundled into a single output value. The return value of the function follows the scheme

$$return = \begin{cases} -1 & \text{if } V = (0,0) \\ 0 & \text{if } ucz_1 \\ n & \text{otherwise } dz_n \text{ with } n > 0 \end{cases}.$$

Instead of ∞ the implementation uses a hardcoded 99 in case of the agent leads to the goal by failures. The 99 was chosen because it is many times larger than the possible degrees of the DZ, which are limited by the height and width of the map. At the same time, 99 is not so much larger that it skews the loss too much. The return array is stored in the replay buffer as an additional feature and is used for the DZ_UCZ task.

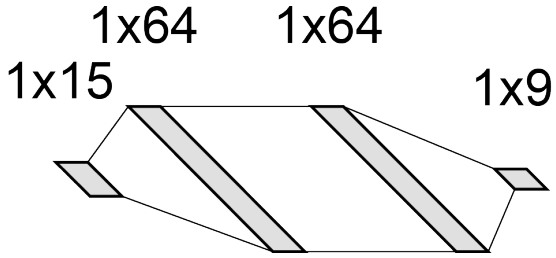


Figure 6.5.: Basic DZ_UCZ network adaption for *Racetrack* [1].

The neural network DZ_UCZ: $\mathbb{R}^{15} \to \mathbb{R}^9$ uses the same input and hidden layer as the other auxiliary tasks but uses a nine neuron output layer and predicts the resulting DZ and UCZ for all possible nine actions. This structure is depicted in Figure 6.5.

Since data from simulations is used, the loss function, in this case, is a mean squared error over the entire array of the form

$$\mathcal{L}_{DZ\_UCZ} = \mathbb{E}_{(s,dz\_ucz)\sim\mathcal{U}(B)}\left[\|DZ\_UCZ(s,\theta) - dz\_ucz\|_2^2\right]$$

where $DZ\_UCZ(s;\theta)$ is the predicted DZ and UCZ array from the neural network and dz_ucz is the observed DZ and UCZ from the environment stored in the replay buffer.

6.3. Training the Agents

The different tasks can be combined into different agents. The more tasks are combined, the larger the required neural network becomes and the bigger the interference between

the tasks gets. However, more tasks also help to understand more aspects of the environment [22]. In the following the used set of agents is explained and how they were trained on the *Racetrack*.

6.3.1. Task Combination

With one main A2C task and four auxiliary tasks, there are $2^4 = 16$ different combinations of agents, calculated from the power set from none to all auxiliary tasks. Since this amount of agents would have exceeded the capacity of this work, the experiments were carried out with a subset, which was composed as follows:

- A2C: The pure A2C
- UNREAL: The UNREAL adaption combining A2C with FP, RP and VR
- A2C_VR: The A2C combined with VR
- A2C_DZ_UCZ: The A2C combined with DZ_UCZ
- A2C_VR_DZ_UCZ: The A2C combined with VR and DZ_UCZ

To make a fair comparison, all approaches use the same fully connected feed-forward network architecture.

In addition, a DQN version [12] was adopted as a reference to having a comparison with other DRL approaches. This approach has proven successful on the *Racetrack* and can be understood as an upper bound [12].

6.3.2. Hyperparameters

The overall loss function

$$\mathcal{L}(\theta) = \lambda_\pi \mathcal{L}_\pi + \lambda_v \mathcal{L}_v - \mathbb{E}_{s\sim\pi}[\lambda_H H(\pi(s;\theta))] + \lambda_{VR}\mathcal{L}_{VR} + \lambda_{FP}\mathcal{L}_{FP} + \lambda_{RP}\mathcal{L}_{RP} + \lambda_{DZ\_UCZ}\mathcal{L}_{DZ\_UCZ}$$

has seven weights, which could assign differently and set the focus on different tasks. An exhaustive hyperparameter tuning is out of the scope of this work. Therefore, prior knowledge and simplifications are used.

First of all the weights λ_π, λ_v, λ_{VR}, λ_{FP}, λ_{RP} and λ_{DZ_UCZ} are set fixed to 1. For λ_π,

λ_v, λ_{VR} and λ_{RP} this was the setting from Jaderberg et al. [22] in their experiments. In the case of the λ_{FP} task, the task has been modified so much that it does not seem appropriate to continue using the original weights. Therefore, this task is also set to 1. In the case of the λ_{DZ_UCZ} task, no prior knowledge exists, so the same weight is assumed for this task as for all others.

The entropy has an impact as illustrated in Figure 3.6b. The loss sum can vary greatly depending on the number of auxiliary tasks. Therefore, the entropy regularization weight λ_H is tuned over an array of the form $(0.005, 0.01, 0.02, 0.04)$, independently for each algorithm.

The replay buffer size is set to 2000, the same size that Jaderberg et al. [22] used. They gave no specifications regarding the batch size. Since the batch size should not be too large to avoid too frequent cancellations of the RP task, it is set to 64. This allows an average episode length of 31 steps to have enough non-zero rewards for a skewed sampling.

Furthermore, the 20-step return is reduced to a 4-step return, as the episodes on the *Racetrack* environment are very short, especially with a bad policy.

For the learning rate, the value of $5 \cdot 10^{-4}$ is chosen. The discount factor γ is fixed at 0.95, which has given good results in experiments with the DQN [12].

The DQN agent uses the same learning rate, discount factor, batch size and buffer size. To avoid possible performance losses due to a higher update rate, it was trained with an update every four environment steps as well as an update every 20 environment steps. The other hyperparameters are set according to prior work [12].

6.3.3. Training

The training is performed on the maps Barto-small, visualized in Figure 4.2a, Barto-big, visualized in Figure 6.6, and River-deadend, visualized in Figure 6.7, with all agents and all four possible *Racetrack* modes, as explained in section 4.2. All agents are trained using the same number of training episodes.

The training progress in RL is strongly influenced by the random seed. To better control this influence, all agents are trained with six different seeds from 0 to 5. The length of an episode was set to 100 and the number of episodes in training to 20000 for the maps Barto-small and Barto-big. For the River-deadend map these two parameters are doubled. The policy was stored every 1000 episodes, as well as the best policy, chosen by the best mean reward in 100 episodes, and the final policy, which in total leads to 22 policies per training run. For the River-deadend map only every second stored policy is considered for evaluation, which keeps the number of policy per agent equal.

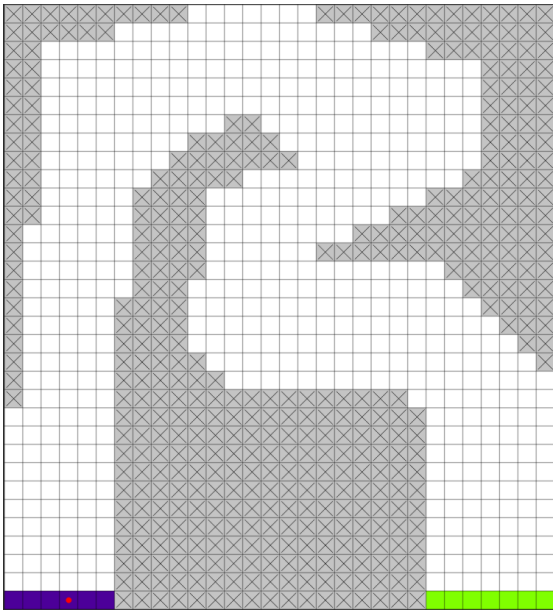


Figure 6.6.: *Racetrack* map with name Barto-big [18].

In the end this results in $|\text{Modes}| \cdot |\text{Agents}| \cdot |\text{Entropy}| \cdot |\text{Seeds}| \cdot |\text{Policies}| = 4 \cdot 5 \cdot 4 \cdot 6 \cdot 22 = 10560$ agents for the final evaluation per map. Additionally also 1056 DQN agents were evaluated.

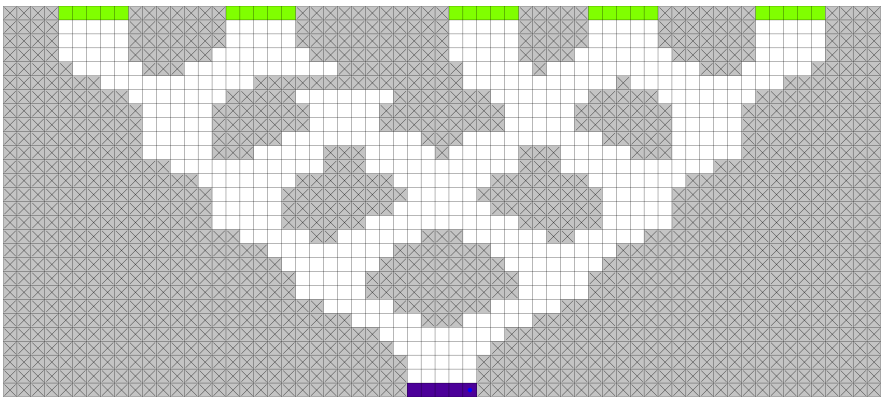


Figure 6.7.: *Racetrack* map with name River-deadend [18].

7. Empirical Evaluation

All the approaches are compared using the measures proposed by Jaderberg et al. [22]: robustness of training, speedup of learning progress, and performance. The results of these measures are reported in the first part of this chapter. As explained in section 6.3 there are 10560 policies per map to evaluate. These are 528 policies for each agent on each map. For the evaluation Deep Statistical Model Checking (DSMC) [14] is used.

DSMC is a straightforward variation of statistical model checking and uses a neural network to resolve the non-determinism in a MDP. This results in a Markov chain, which can be analysed statistically and verify properties of the neural network, such as the mean return and the goal reachability probability in case of *Racetrack* [14]. For the statistical evaluation in this work, the error bound $P(error > \epsilon) < \kappa$ is set $\kappa = 0.05$ and $\epsilon = 1$ for the mean return and $\epsilon = 0.05$ for the goal reachability probability, i.e. a confidence of 95% that the mean return is in an interval smaller than 1 and the goal reachability probability in an interval smaller than 0.05.

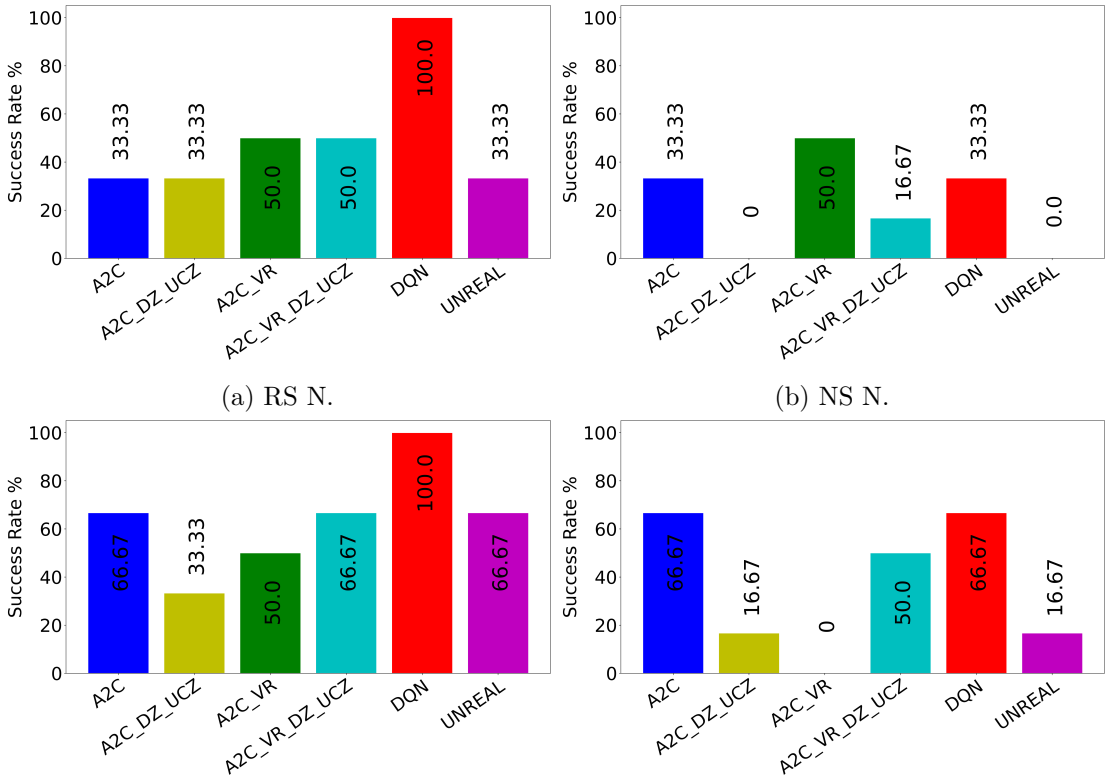
The reported results always refer to the policy with the highest goal reachability probability. For the evaluation, the agent's policy is not probabilistic but deterministic, and it only follows the action with the highest probability. The auxiliary tasks do not play a role in the evaluation.

The section 7.4 of this chapter takes a closer look at the behaviour of the different agents and how the DZ_UCZ tasks influence this behaviour.

7.1. Robustness of Training

To measure the robustness of training, the best results across all seeds are used for the best entropy weighting factor. The best entropy weighting factor is determined by the policy with the highest goal reachability probability. A training run is considered as successful if the resulting policy can solve at least 50% of the episodes used for evaluation.

The results for the Barto-small map, illustrated in Figure 7.1, visualize the percentage of successful training runs for all four *Racetrack* modes. In the N modes, shown in the top line, the auxiliary task shows a benefit over the A2C. In the NS D mode, on the bottom right, the A2C performs better without auxiliary tasks. Especially in the RS modes, the DQN is more robust compared to the other agents. In the NS modes the policy-based approaches are equal or as in the NS N mode on the top right more robust.



(a) RS N.

(b) NS N.

(c) RS D.

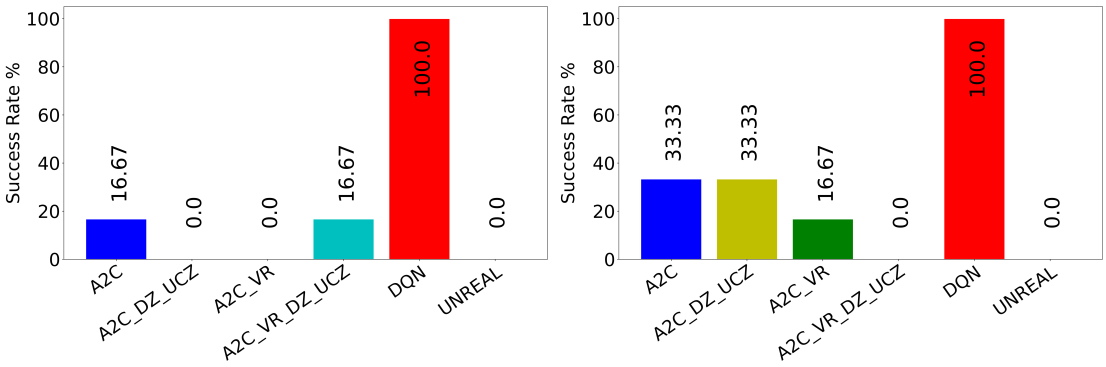
(d) NS D.

Figure 7.1.: Robustness of training Barto-small.

In Figure 7.2 the results for the Barto-big map are presented. The NS modes are not solved by any approach, including the DQN, therefore no graphics were created for it. This suggests that these modes were too difficult to solve with the chosen approaches. The UNREAL never reaches the goal in all modes, while all approaches with auxiliary tasks only perform well in one mode. It seems that this map is simply too difficult for the approaches chosen, except for the DQN which is robust in both RS modes.

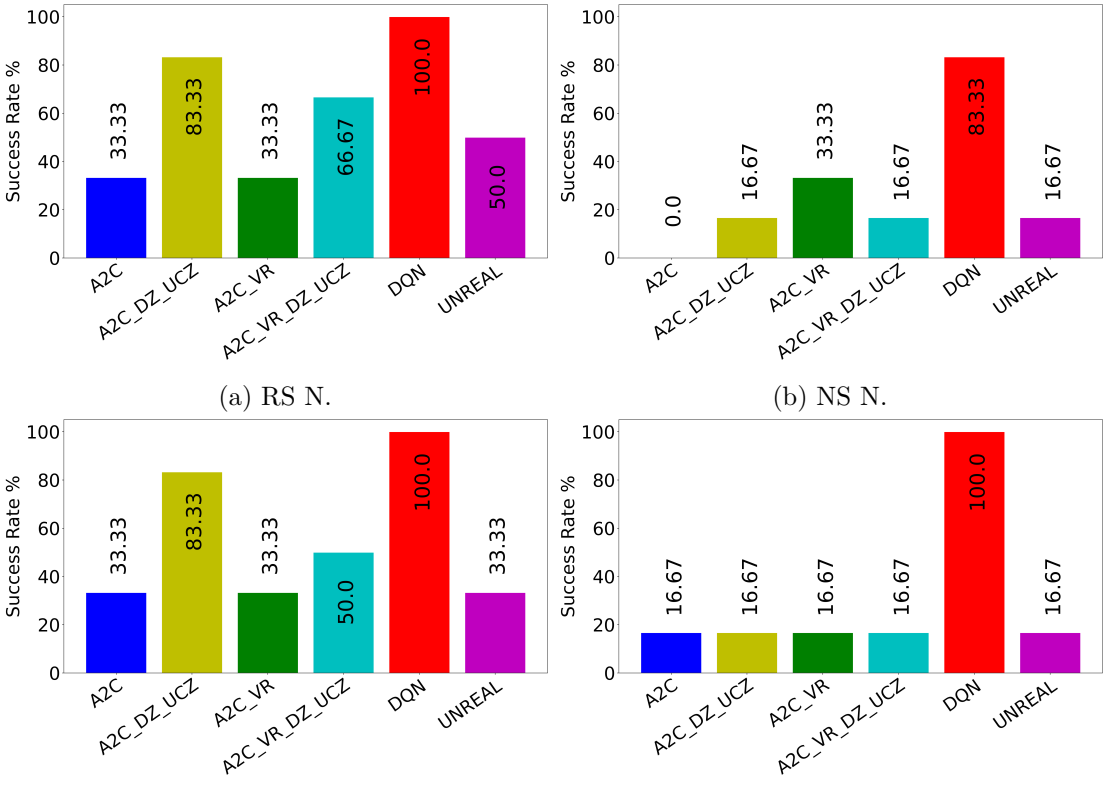
The results on the River-deadend map illustrated in Figure 7.3, show the benefit of DZ_UCZ tasks as in the RS modes the algorithms with DZ_UCZ tasks are more robust compared to A2C and more stable than other auxiliary tasks, including UNREAL, and come much closer to the robustness of DQN than in NS modes. Apart from the NS N mode, the DQN has maximal robustness.

It can be stated that the auxiliary tasks have a stabilizing effect, the DZ_UCZ tasks, especially in N and RS modes. The DQN as upper bound was exceeded only once. The DQN proved to be the more robust approach in the RS modes.



(a) RS N. (b) RS D.

Figure 7.2.: Robustness of training Barto-big.



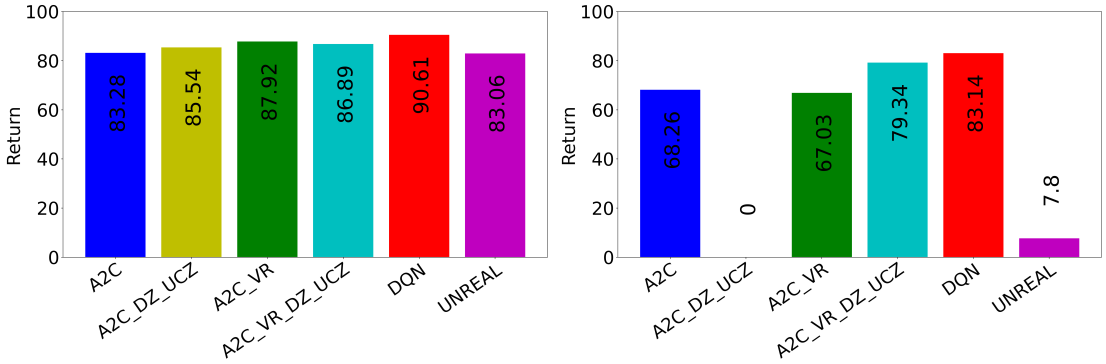
(a) RS N. (b) NS N.

(c) RS D. (d) NS D.

Figure 7.3.: Robustness of training River-deadend.

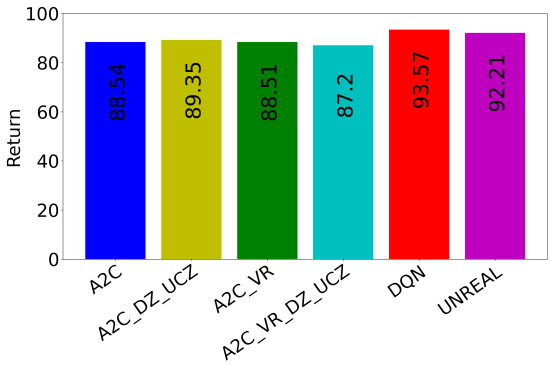
7.2. Performance

To measure the performance of the resulting agent, the achieved return and the goal reachability probability are considered. The latter is especially meaningful for evaluation, as the primary goal of the course is not to crash while still reaching the goal. Further, the construction of a DZ and UCZ should primarily lead to safer, not faster driving.

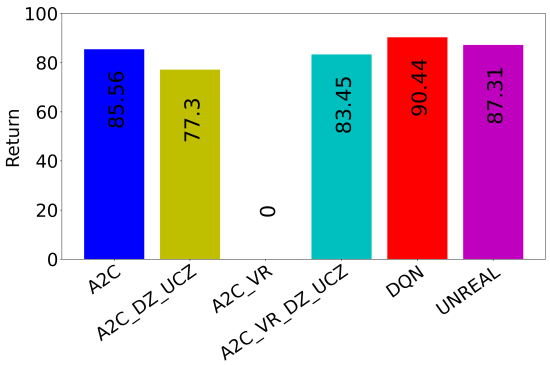


(a) RS N.

(b) NS N.



(c) RS D.

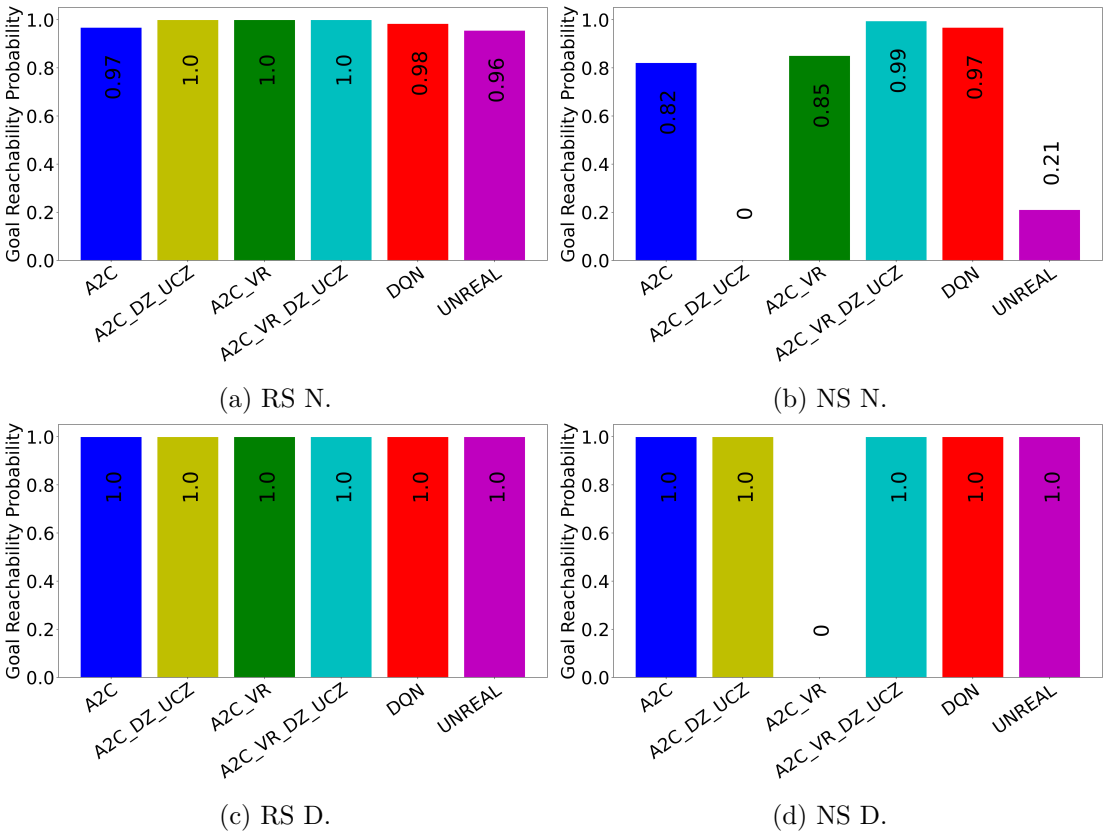


(d) NS D.

Figure 7.4.: Return Barto-small.

In Figure 7.4 the best-achieved returns are presented for the Barto-small map. All agents receive a nearly equal maximum return, except for three outliers in the NS modes on the right. The DQN achieves the maximum reward in all modes. Since there are near equal returns in the RS N and RS D modes on the left, there is no outstanding difference between the approaches and therefore no way to tell if one has an advantage over the other. In the D modes, the UNREAL achieved the second maximum return. On the other hand, in the N modes, the UNREAL achieved the minimal return. The A2C_VR_DZ_UCZ approach achieved a significantly higher return in the NS N mode compared to the other policy-based approaches.

These findings are confirmed by looking at the goal reachability probability. In the RS D mode, see Figure 7.5c, all approaches reach the goal in all episodes. Also in the NS D mode, see Figure 7.5d, all approaches, except A2C_VR, reach the goal successfully.



(a) RS N.

(b) NS N.

(c) RS D.

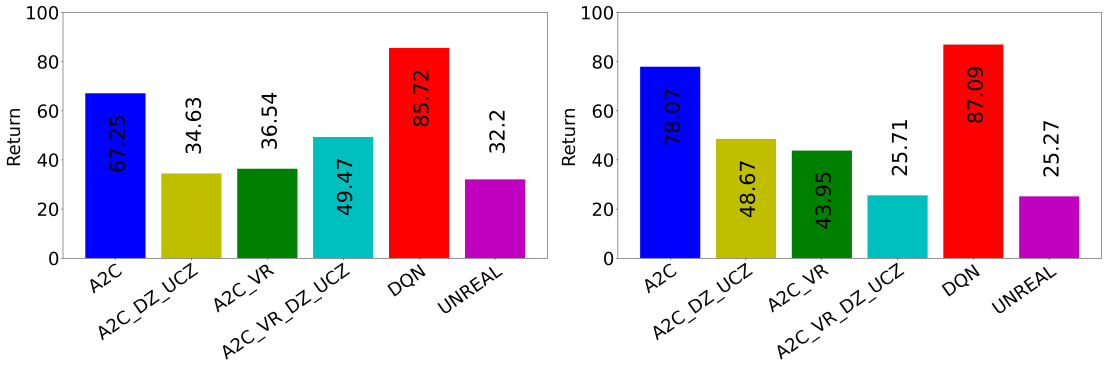
(d) NS D.

Figure 7.5.: Goal reachability probability Barto-small.

In the N modes, the approaches using DZ_UCZ tasks outperform the DQN. This is impressive since the D modes all perform equally well. The higher goal reachability probability and the lower return lead to the assumption that the approaches with DZ_UCZ tasks need more steps to the goal, which means driving safer. This seems to be beneficial in the N modes, as they have a higher goal reachability probability.

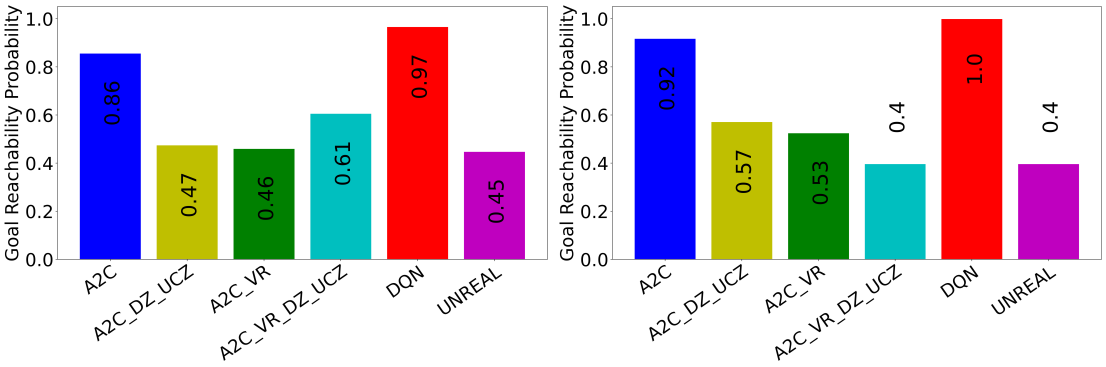
The maximum return for the Barto-big map leads to the same interpretation as the robustness. The returns are mapped in Figure 7.6. No agent reaches the goal in the NS modes. As the return is zero, this is evidence that the agents have all learned to sit out an episode by standing still to avoid crashing. In the RS modes, DQN and A2C outperform the approaches with auxiliary tasks. UNREAL is the worst performing approach in both modes.

The goal reachability probability for the Barto-big map confirms this. The DQN has perfect goal reachability probability in the RS D mode and slid less in the RS N mode. The bad performance of the auxiliary task could be a result of high interference by the different tasks. Therefore, this map is treated as an outlier for further analysis. Further experiments may provide insight into why this map performs so much worse, but this would be beyond the scope of this work.



(a) RS N. (b) RS D.

Figure 7.6.: Retrun Barto-big.

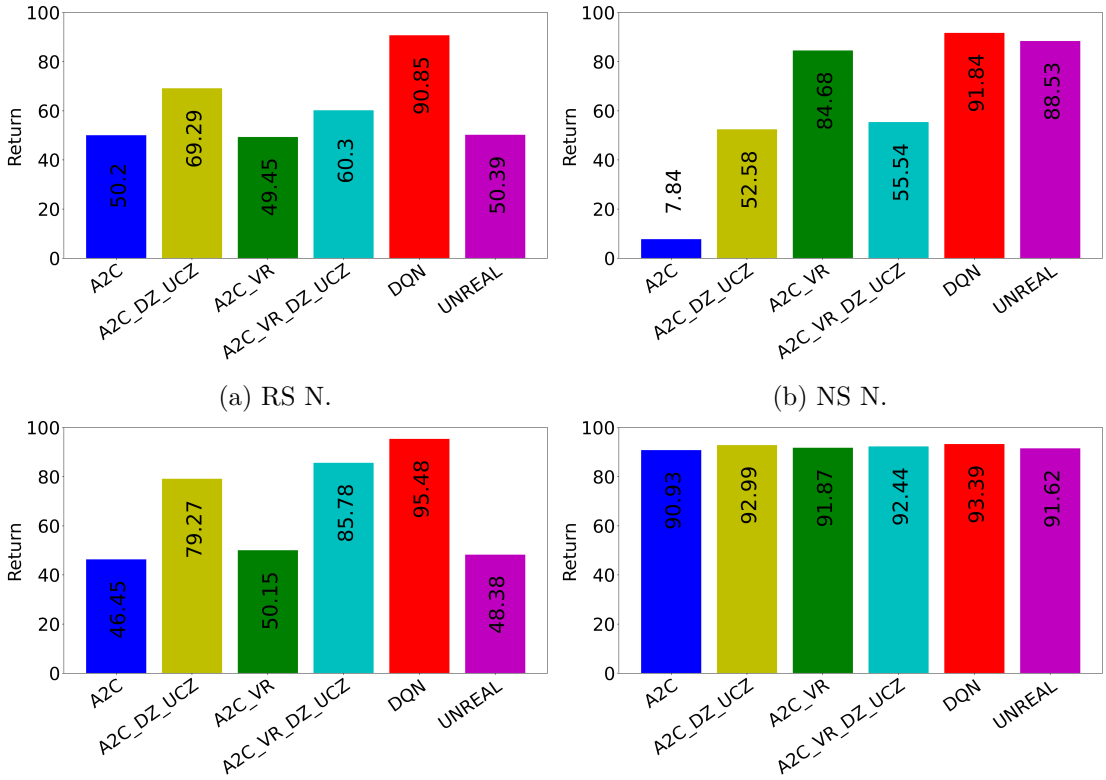


(a) RS N. (b) RS D.

Figure 7.7.: Goal reachability probability Barto-big.

The returns for the River-deadend map confirm the usability of the DZ_UCZ tasks. The returns are shown in Figure 7.8. It is noteworthy that in the RS modes on the left, the approaches with DZ_UCZ tasks achieved a significantly higher return compared to the other policy-based approaches. A2C_DZ_UCZ and A2C_VR_DZ_UCZ clearly outperform A2C and UNREAL. In the NS N mode on the top right, the UNREAL approach achieves a near equal return compared to the DQN. In the NS D mode, all achieved a near equal return.

The goal reachability probability for the River-deadend map, illustrated in Figure 7.9, confirms these findings. In the NS D mode, on the bottom right, all approaches reach the goal in all episodes. In the RS modes, on the left, the approaches with DZ_UCZ tasks reach the goal significantly more often than the other approaches, except the DQN, which performs near perfect over all four modes. In the NS N mode, the UNREAL approach achieved the second-best goal reachability probability.

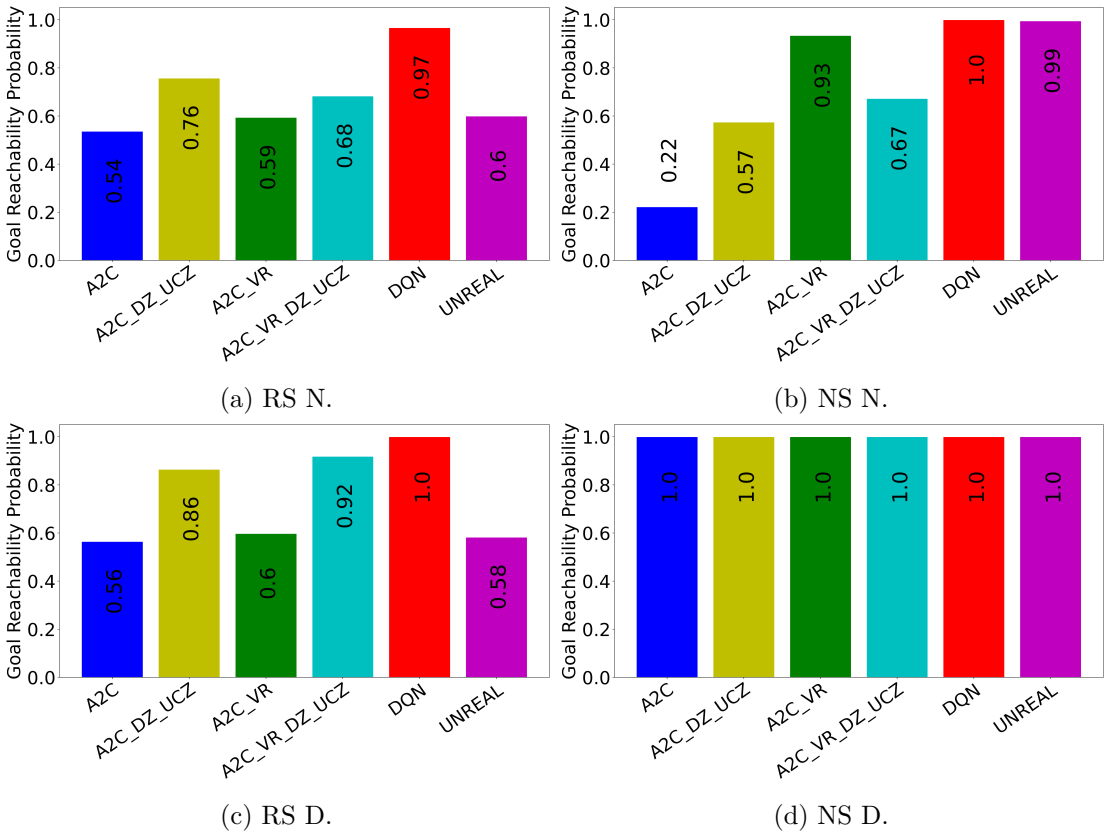


(a) RS N. (b) NS N.

(c) RS D. (d) NS D.

Figure 7.8.: Return River-deadend.

All these findings lead to the result that the DZ_UCZ tasks are beneficial, especially in RS N modes, by increasing the goal reachability probability through safer driving, as indicated by the partially lower return.



(a) RS N. (b) NS N.

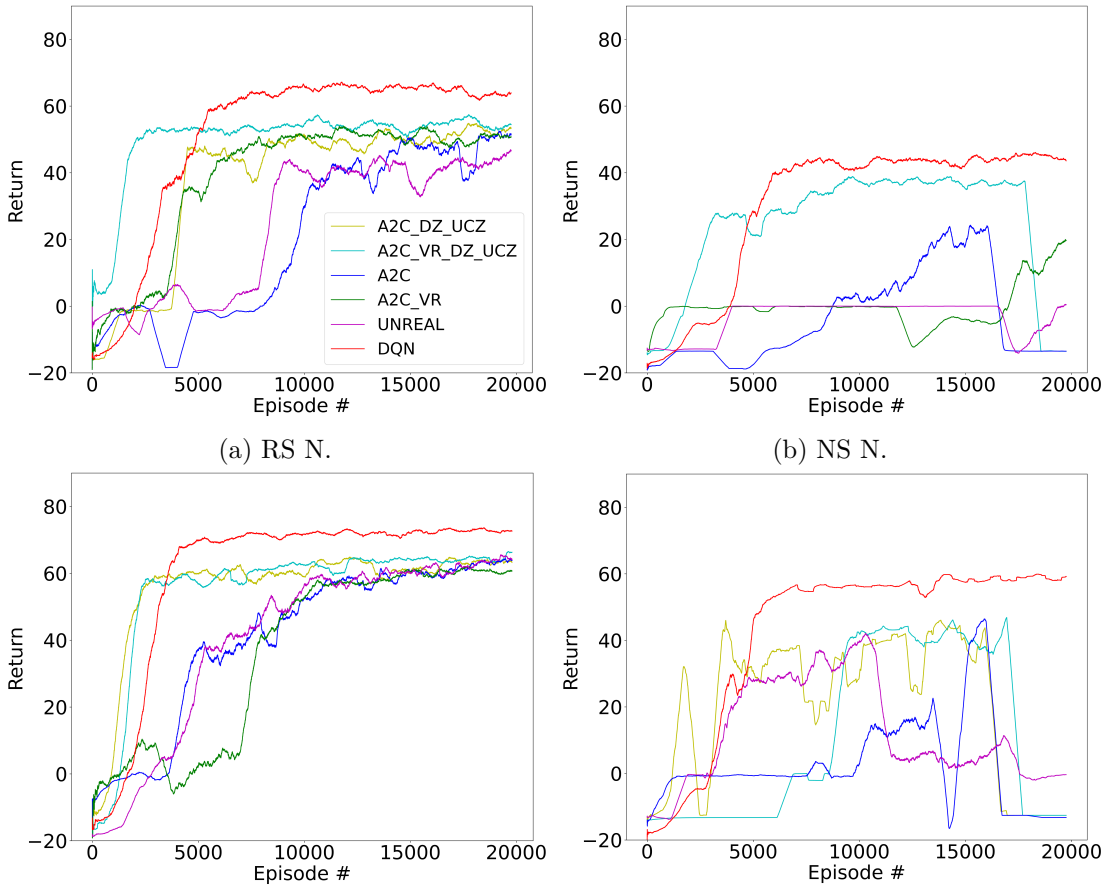
(c) RS D. (d) NS D.

Figure 7.9.: Goal reachability probability River-deadend.

7.3. Learning Progress Speed

To measure the learning speed, the learning curves are considered. The results for the Barto-big map were excluded because the results in the previous analyses were too weak. Therefore, only the learning curves of Barto-small and River-deadend are considered, using the learning curves of the best agents from the analyses before. Note that the best policy is not necessarily the end policy, since all 1000 steps intermediated policies are stored and evaluated. During the training, the agent can overlearn and therefore the performance can collapse. Approaches that never reach the goal in the evaluation are excluded.

Figure 7.10 provides the learning curves for the Barto-small map on all four modes. DQN (red) learns best on all four modes, by achieving a higher return. Both agents using the DZ_UCZ task (cyan and yellow) do learn faster compared to other policy-based approaches in the RS modes, on the left, as can be seen by the faster increase of the training curve. They also learn faster compared to the DQN as they reach their maximum return in fewer steps. This is particularly pronounced in the RS N mode at the top left.



(a) RS N.

(b) NS N.

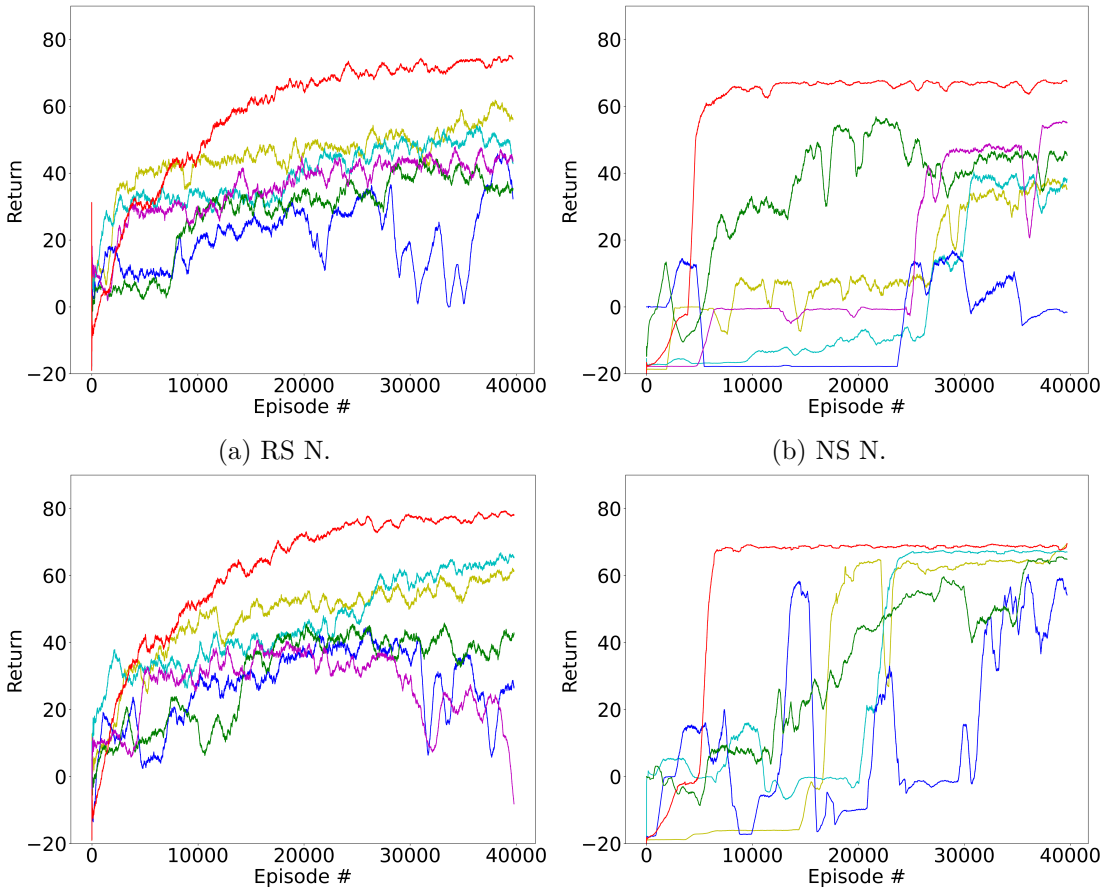
(c) RS D.

(d) NS D.

Figure 7.10.: Learning Progress Barto-small.

However, it is important to note that the DQN uses a random policy with decreasing probability for exploration, which makes the comparison difficult. The basic A2C learns slowest in the N modes. The NS D mode, on the bottom right, is quite confusing as all approaches, except DQN, are not stable in performance improvement.

The learning curves for the River-deadend map, shown in Figure 7.11, imply the same results. The DQN learns best for all four modes. Both agents using the DZ_UCZ task (cyan and yellow) do learn faster compared to other policy-based approaches in the RS modes on the left. In the NS N mode, on the top right, the A2C_VR learns faster than other policy-based agents. For all modes, A2C without any auxiliary task is learning the slowest. All these results show that the DZ_UCZ speedup the learning progress, especially in the RS and N modes.



(a) RS N.

(b) NS N.

(c) RS D.

(d) NS D.

Figure 7.11.: Learning Progress River-deadend.

7.4. Policy Behaviour

To examine the policy behaviour in the Barto-small and River-deadend map more closely, the paths were tracked in 1000 played episodes. For this purpose, the policy with the highest goal reachability probability was used. After each episode, the visited states were analyzed and the degrees of DZ and UCZ were determined. The first and the last state were ignored because they are always a state with $V = (0, 0)$ and a terminal state. Therefore, these states are equal for all approaches and not useful to determine differences. Timeout episodes are not taken into account when calculating the frequencies, as these would strongly distort the results.

Only the N modes were considered because there the DZ_UCZ tasks show a performance improvement and the agent might have learned in training to avoid DZ with a low degree. Also, the N modes are more interesting for analysing behaviour because the agents need to deal with noise and therefore adapt its behaviour.

The frequency of each DZ was counted for all episodes and divided by the number of episodes. This results in an average frequency with which the agent stays in a danger zone per episode. This gives information about the total length of a mean episode, as well as about the agent's willingness to take risks. The UCZ also provides information about how often a crash was no longer avoidable, which means that the agent was in an unsolvable situation.

7.4.1. Barto-small Map

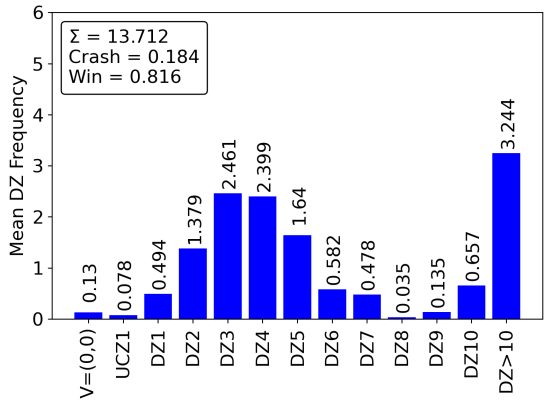
Figure 7.12 shows the results for all approaches in the Barto-small map with the NS N mode. In the top left corner of each plot, the average number of steps per episode as well as the win and crash rates are shown. The A2C_DZ_UCZ is not illustrated, because it never reaches the goal.

What can be seen is that the approaches using the DZ_UCZ tasks need more steps, but have a lower proportion of DZ with a low degree, which suggests a safer driving behaviour. Also, the number of steps of stop and go, expressed through the bar $V = (0, 0)$ is much higher, which confirms this assumption. The A2C_VR_DZ_UCZ has a 78 times lower rate of UCZ compared to the A2C and also a 22 times lower rate than the DQN. At the same time, on average, it needs about 9 steps more than the A2C. Of these, almost 3 are spent on stop and go, which is a waste of steps, but a safer driving behaviour. The DQN has the least amount of stop and go of all approaches that do not timeout, so it does not waste steps. Noteworthy is the number of UCZ into which the UNREAL agent runs, which means that almost half of all crashes were unavoidable.

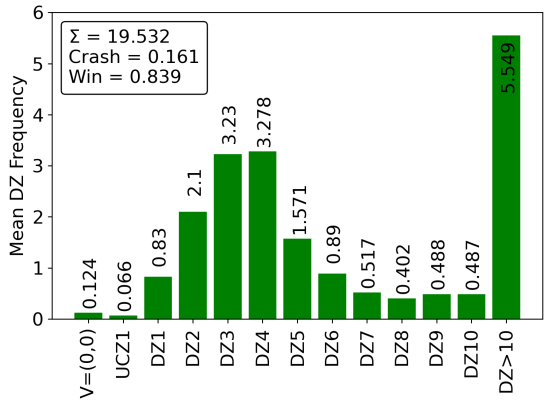
Appendix A.1 includes illustrations of the different agents as they travel a sample path. This illustration shows that the A2C_VR_DZ_UCZ approach is manoeuvring very slowly and in a straight line. On the other hand, both the DQN and the A2C try to drive parabolically towards the goal, because this is the shortest path. The A2C_VR, on the other hand, drives very chaotically.

Figure 7.13 shows the results for the same agents again but this time in the RS N mode. One can see that the RS N mode is simpler than the NS N mode, which is why all approaches have a high goal reachability probability. At the same time, the rate of UCZ is very low. It is nevertheless easy to see that the A2C_VR_DZ_UCZ, shown in Figure 7.13d, has a very low rate in low DZ. The proportion of DZ_1 is 6 times lower than in DQN and 14 times lower than in A2C. The average sum of steps is near equal, apart from the DQN which needs on average 40% fewer steps compared to the second-best approach. Again, this is illustrated in appendix A.2.

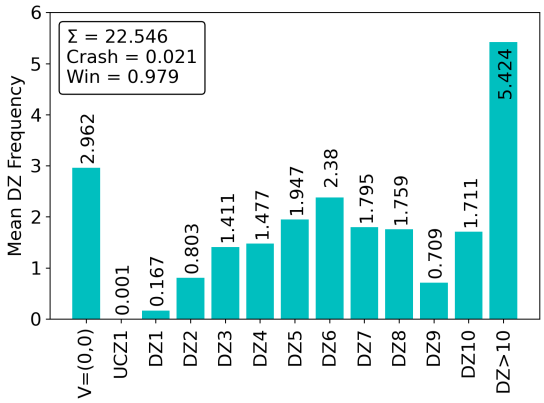
It can be stated that the DZ_UCZ tasks lead to a safer driving style, which is proven by the lower proportion of UCZ and a lower frequency of low-degree DZ. The lower frequency can be explained either by a larger distance to the wall or by a lower speed. Both characteristics speak for a safer driving style.



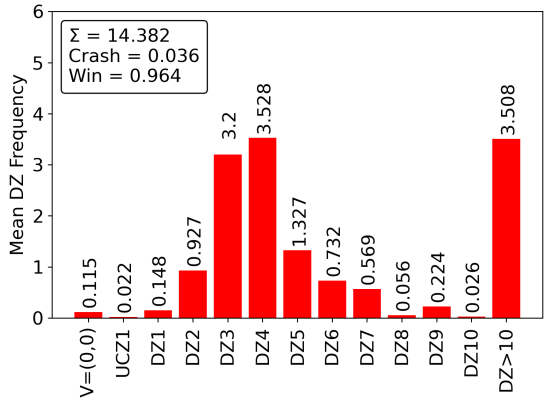
(a) A2C.



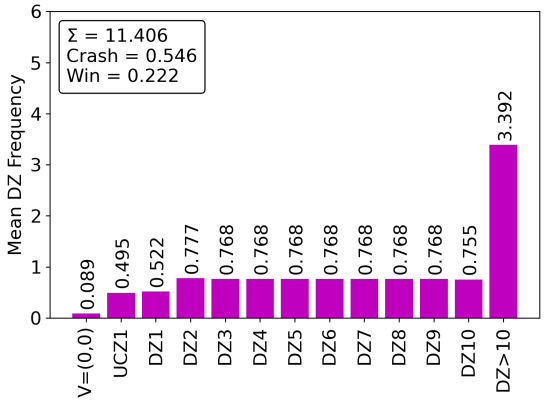
(b) A2C_VR.



(c) A2C_VR_DZ_UCZ.

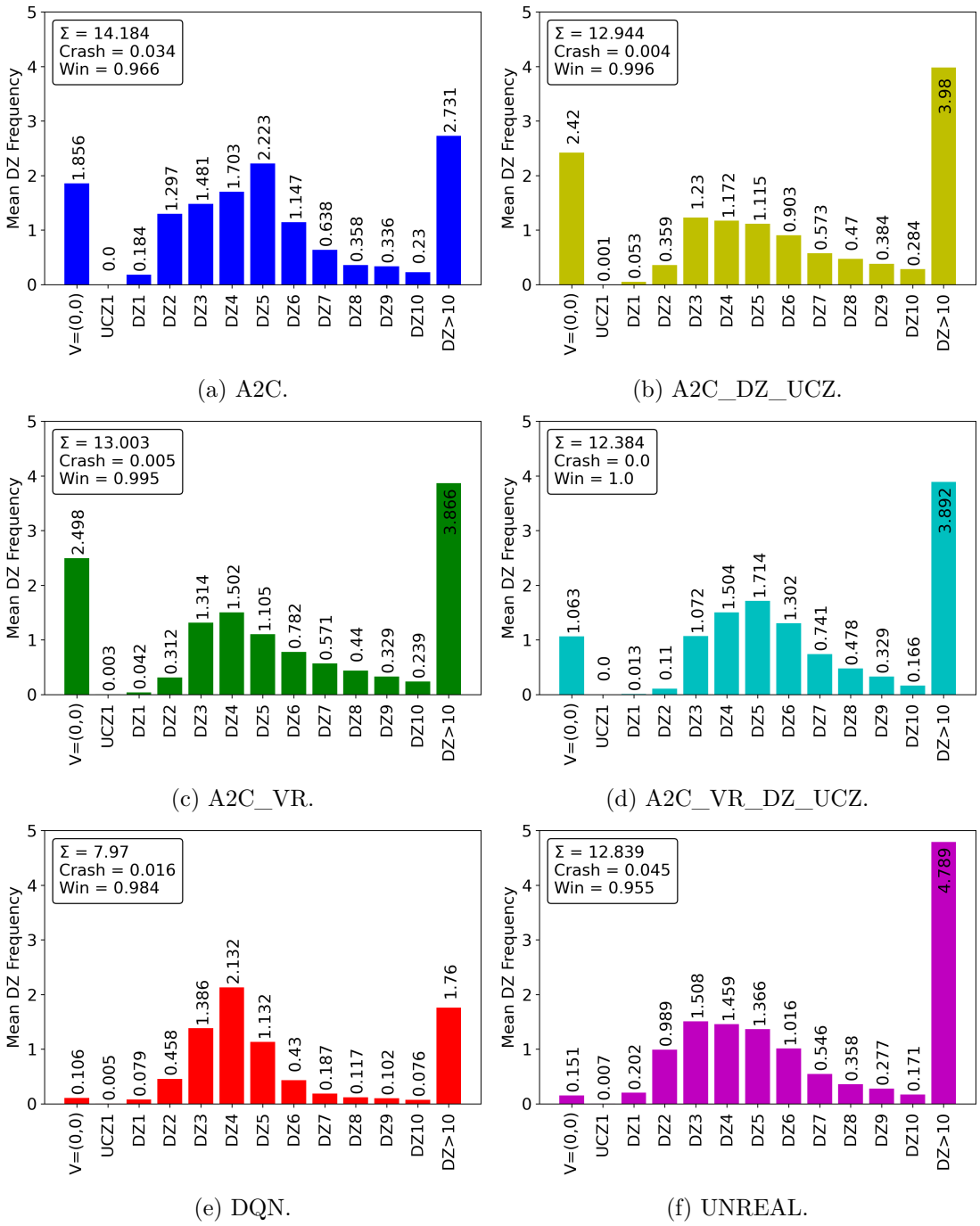


(d) DQN.



(e) UNREAL.

Figure 7.12.: Analyse of visited DZ in Barto-small map in NS N mode.



(a) A2C. (b) A2C_DZ_UCZ.

(c) A2C_VR. (d) A2C_VR_DZ_UCZ.

(e) DQN. (f) UNREAL.

Figure 7.13.: Analyse of visited DZ in Barto-small map in RS N mode.

7.4.2. River-deadend Map

Figure 7.14 illustrates the agents on the River-deadend map in the NS N mode. Since the map is more complicated this time, there is a higher rate of timeouts. This higher rate is need to be considered since this reduces the mean frequency of all zones.

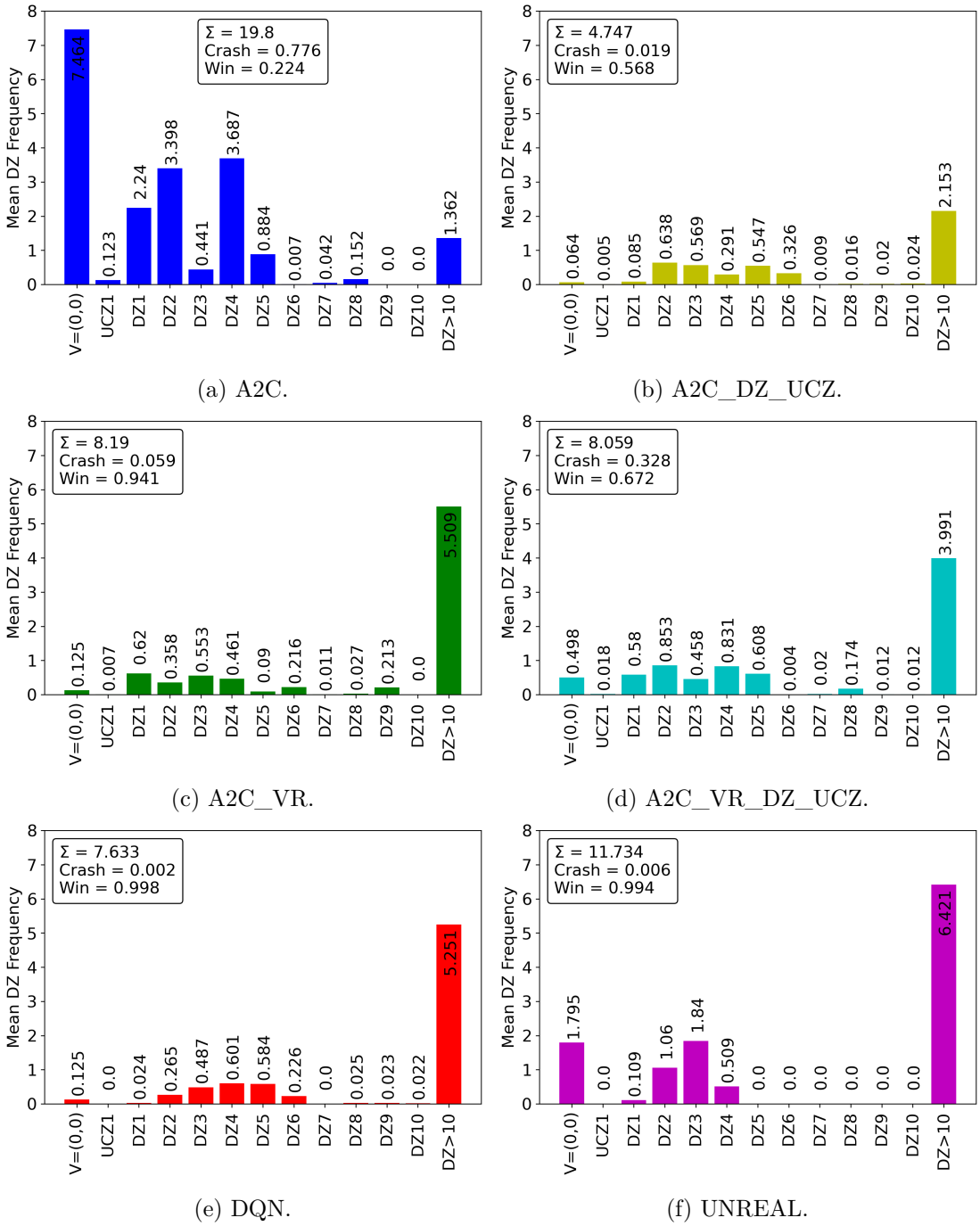
The agents A2C, A2C_VR A2C_VR_DZ_UCZ, DQN and UNREAL reach the goal or crash at 100% of all episodes, while agent A2C_DZ_UCZ timeouts in near 40% of all episodes. The DQN agent performs best with the lowest frequency of UCZ and DZ_1 and the lowest number of steps. The UNREAL also performs with zero frequency of UCZ but has a higher frequency in lower degrees of DZ. The A2C has the highest frequency in UCZ as well as DZ with lower degrees.

As the illustrations in appendix A.3 visualize, the high performance of the A2C_VR and UNREAL approaches leads to the fact that there is an easy way to reach the goal. The agents drive a direct path to the upper corners. These paths have a low risk, which can be seen in the high degree of DZ. The frequency of high DZ is also increased for these two agents, which can be seen in Figure 7.14c and Figure 7.14f. Since the approaches with the DZ_UCZ task have a significantly lower winning rate, likely, they do not use this simple path.

Figure 7.15 illustrates the same approaches for the RS N mode. There is no straight line solution in this case, making this mode more complex to manoeuvre. Also in this mode the A2C agent times out in around 35% of the episodes, which reduces the number of steps. Additionally, the A2C_DZ_UCZ times out in nearly 5% of the episodes. The A2C_DZ_UCZ and A2C_VR_DZ_UCZ have a higher goal reachability probability and slide lower frequency of UCZ compared to the other policy-based approaches.

The DQN outperforms all other agents. It has the lowest number of steps, the highest goal reachability probability and the lowest frequency of UCZ and DZ_1 In this mode, the A2C_VR and UNREAL approaches no longer have outstanding performance, which confirms the assumption that the path was very easy to find in the NS N mode, but no longer in the more complex RS N mode. For this mode, there are also some illustrations in the appendix A.4.

To summarize, it can be said that the DZ_UCZ tasks also lead to safe driving behaviour in the River-deadend map. However, this task also seems to prevent easy solutions from being recognized directly. There is still room for improvement.



(a) A2C.

(b) A2C_DZ_UCZ.

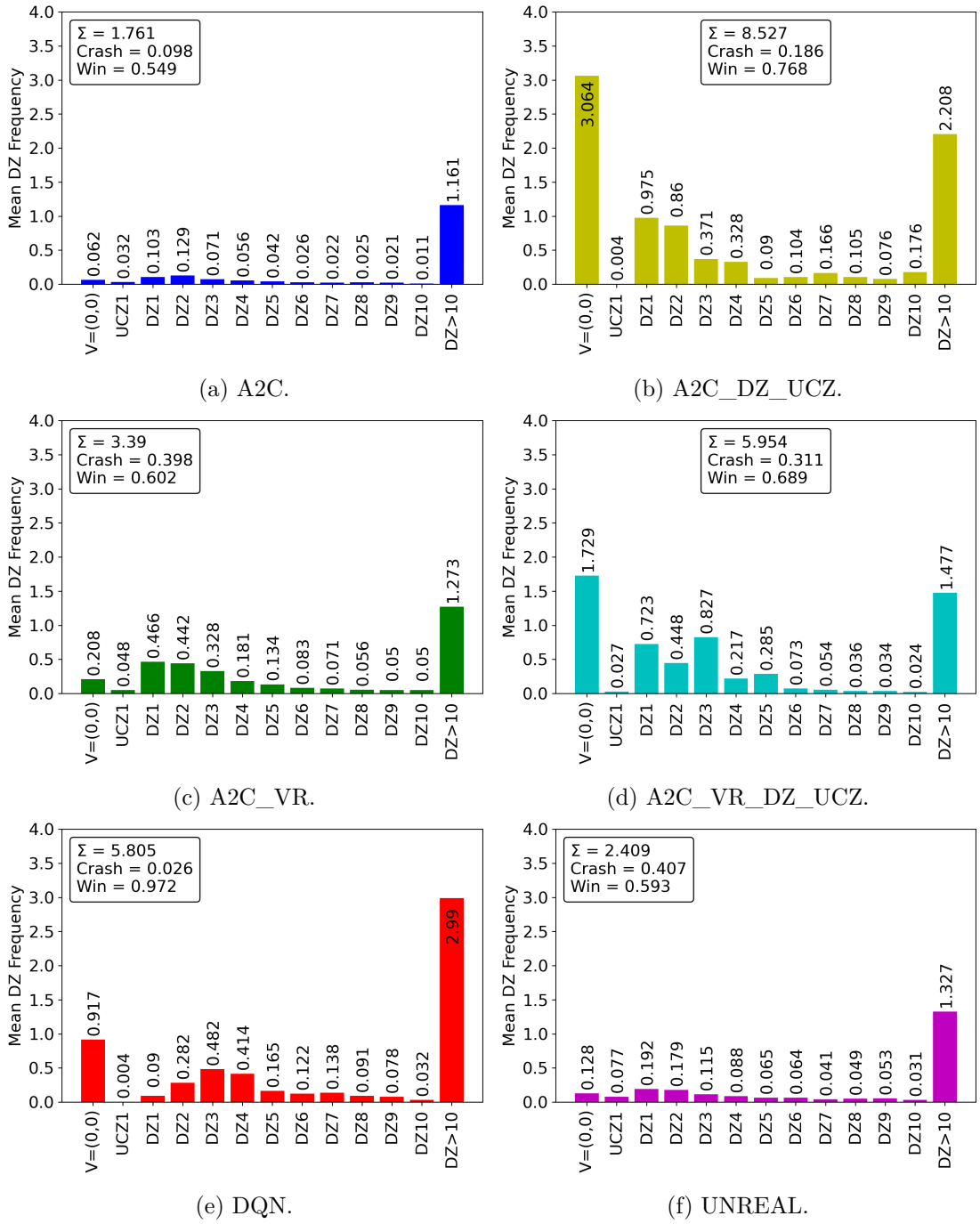
(c) A2C_VR.

(d) A2C_VR_DZ_UCZ.

(e) DQN.

(f) UNREAL.

Figure 7.14.: Analyse of visited DZ in River-deadend map in NS N mode.



(a) A2C. (b) A2C_DZ_UCZ.

(c) A2C_VR. (d) A2C_VR_DZ_UCZ.

(e) DQN. (f) UNREAL.

Figure 7.15.: Analyse of visited DZ in River-deadend map in RS N mode.

8. Conclusion

8.1. Recapitulation

In this work, the usability of LTL and CTL encoded auxiliary tasks for deep reinforcement learning is explored. Appropriately designed auxiliary tasks improve the dynamics and stability of representation learning, which has been shown to be particularly useful in sparse-reward settings. The problem is that there is currently no general method for creating good auxiliary tasks and a clear definition of what constitutes a good auxiliary task is still missing. Good auxiliary tasks should allow for an improvement of representation learning in a majority of different environments and should not be too specific, in order to be generally applicable to many different environments with different tasks and input types. At the same time, they should not be too generic to convey an understanding of the behaviour that the individual tasks learn.

Temporal logic provides a reasonable framework for a more general view of auxiliary task design that is not limited to a particular class of benchmarks. This work proposes a systematic way for generating tasks that reason about the local environment of the agent. The tasks used in this work assume a minimum amount of atomic propositions, which can be generalized to many benchmarks. The tasks are beneficial during the early exploration phase when the reward signal is weak or missing as they provide additional information for the behaviour, independent of the reward structure. In this work, two CTL encoded auxiliary tasks were presented. For one of them, a simplified but *Racetrack* specific LTL version was presented, which expresses the same property.

The empirical evaluation using the *Racetrack* shows that the agents using temporal logic encoded auxiliary tasks perform better compared to A2C on two of three maps. Moreover, in the N modes of the *Racetrack*, the agents using temporal logic encoded auxiliary tasks outperform the A2C as well as the adaption of UNREAL considering robustness of training, learning speed and performance. In one setting, the temporal logic expanded A2C agent even outperforms the value-based approach of DQN, which is known to perform better than policy-based approaches on the *Racetrack*. In the N mode on the Barto-small map, the agents using temporal logic encoded auxiliary tasks even outperform the DQN, which is more suited for the *Racetrack*, considering goal reachability probability.

Furthermore, it was investigated how the use of the temporal logic encoded auxiliary tasks affects the behaviour of the agents in the N modes. It could be shown empirically that the temporal logic encoded auxiliary tasks lead to the agents behaving more safely and directing their focus more towards achieving the goal and less on maximizing the reward. Among other things, this leads to the agents getting into unsolvable situations

less often and being in situations where their survival depends on random chance. However, this also leads to the agents taking more steps. This behaviour is different compared to the behaviour of the DQN, which wants to reach the goal in the fewest steps. This leads to a higher return, but more often to a crash. Since the higher return compensates for this risk, this is the optimal solution for the DQN, whose objective leads it to optimize the overall return. The agents with temporal logic encoded auxiliary tasks use the same reward structure and can practice a safer behaviour.

8.2. Future Work

One line of future work is to replace the binary CTL auxiliary tasks with LTL auxiliary tasks. This is done by removing the path quantifiers and instead determining the probabilities for the occurrence of certain atomic propositions using Monte Carlo. As an example on *Racetrack*, the agent would learn the probability of crashing from a certain state.

To generalise the results found, it is necessary to test them on other benchmarks and check whether they also lead to an improvement there or whether these tasks are specific to *Racetrack*. With the technique presented here, additional auxiliary tasks can be defined.

Since the scope of this work did not allow for an exhaustive hyperparameter tuning, the results could still be optimized by better hyperparameters. Especially in the case of the A2C_VR_DZ_UCZ agent, the weighting between VR and DZ_UCZ tasks could be of interest. Furthermore, only a small set of entropy regularization parameters and not all possible combinations of auxiliary tasks have been tested.

Since in this work the two temporal logic encoded auxiliary tasks were combined into one task for the sake of simplicity, it is not possible to say exactly which of the two tasks is responsible for the improvement. To investigate this further, it would be necessary to implement two separate tasks, along with a higher degree UCZ. This could lead to even more far-sighted behaviour, as unavoidable situations would be recognized earlier.

Another approach would be to question whether A2C was the right baseline for *Racetrack*. The basic idea of auxiliary tasks could also be applied to the DQN by extending the neural network of the DQN to predict the auxiliary tasks. This could lead to an improvement similar to the policy-based approaches.

Additionally, one can also combine this work with further techniques of the field, such as evaluation stages [17], the construction [20] or shaping [8] of reward structures, and also with different exploration techniques, e.g., random network distillation [7]. Lastly, auxiliary tasks may also be used to increase the interpretability of the resulting policies, as the auxiliary task can be helpful to explain the agent's behaviour.

Bibliography

[1] Cnn visualization tool, 2022. available at `http://alexlenail.me/NN-SVG/LeNet.html`.

[2] Ron Amit, Ron Meir, and Kamil Ciosek. Discount Factor as a Regularizer in Reinforcement Learning. In *International Conference on Machine Learning*, pages 269–278. PMLR.

[3] Christel Baier, Maria Christakis, Timo P. Gros, David Groß, Stefan Gumhold, Holger Hermanns, Jörg Hoffmann, and Michaela Klauck. Lab Conditions for Research on Explainable Automated Decisions. In *Trustworthy AI–Integrating Learning, Optimization and Reasoning: First International Workshop, TAILOR 2020*, page 83. Springer Nature.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press.

[5] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to Act Using Real-Time Dynamic Programming. 72(1-2):81–138.

[6] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 12–21.

[7] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation.

[8] Mingyu Cai, Mohammadhosein Hasanbeig, Shaoping Xiao, Alessandro Abate, and Zhen Kan. Modular deep reinforcement learning for continuous motion planning with temporal logic. 6(4):7973–7980.

[9] Falk T Gerpott, Sebastian Lang, Tobias Reggelin, Hartmut Zadek, Poti Chaopaisarn, and Sakgasem Ramingwong. Integration of the a2c algorithm for production scheduling in a two-stage hybrid flow shop environment. 200:585–594.

[10] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 10 2000.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[12] Timo P. Gros. Tracking the Race: Analyzing Racetrack Agents Trained with Imitation Learning and Deep Reinforcement Learning. Master's thesis.

[13] Timo P Gros, David Groß, Stefan Gumhold, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Tracevis: towards visualization for deep statistical model checking. In *International Symposium on Leveraging Applications of Formal Methods*, pages 27–46. Springer.

[14] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep statistical model checking. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 96–114. Springer International Publishing.

[15] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep Statistical Model Checking: A Scalability Study. In *International Conference on Quantitative Evaluation of Systems*. submitted but not published yet.

[16] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Models and Infrastructure used in "Deep Statistical Model Checking". available at `http://doi.org/10.5281/zenodo.3760098`.

[17] Timo P Gros, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. Dsmc evaluation stages: Fostering robust and safe behavior in deep reinforcement learning. In *International Conference on Quantitative Evaluation of Systems*, pages 197–216. Springer.

[18] Timo P. Gros, Daniel Höller, Jörg Hoffmann, and Verena Wolf. Tracking the race between deep reinforcement learning and imitation learning – extended version, 2020.

[19] Joshua Hare. Dealing with Sparse Rewards in Reinforcement Learning.

[20] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-constrained reinforcement learning.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, nov 1997.

[22] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks.

[23] Hung-Chin Jang, Yi-Chen Huang, and Hsien-An Chiu. A study on the effectiveness of a2c and a3c reinforcement learning in parking space search in urban areas

problem. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 567–571. IEEE.

[24] Bilal Kartal, Pablo Hernandez-Leal, and Matthew E Taylor. Terminal prediction as an auxiliary task for deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 38–44.

[25] W. Bradley Knox and Peter Stone. Reinforcement Learning from Human Reward: Discounting in Episodic Tasks. In *2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication*, pages 878–885.

[26] Clare Lyle, Mark Rowland, Georg Ostrovski, and Will Dabney. On the effect of auxiliary tasks on representation dynamics. In *International Conference on Artificial Intelligence and Statistics*, pages 1–9. PMLR.

[27] Mark Ryan Michael Huth. *Logic in Computer Science 2ed.* Cambridge University Press, March 2019.

[28] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, feb 2015.

[31] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[32] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by Playing Solving Sparse Reward Tasks From Scratch. In *International Conference on Machine Learning*, pages 4344–4353. PMLR.

[33] Anton Schwartz. A Reinforcement Learning Method for Maximizing Undiscounted

Rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, volume 298, pages 298–305.

[34] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[35] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. 529(7587):484–489.

[36] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-play. 362(6419):1140–1144.

[37] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, oct 2017.

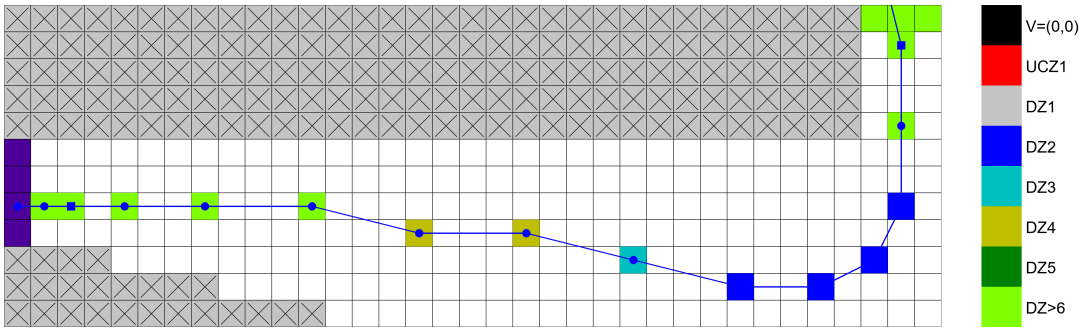
[38] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive computation and machine learning. The MIT Press, second edition.

[39] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768.

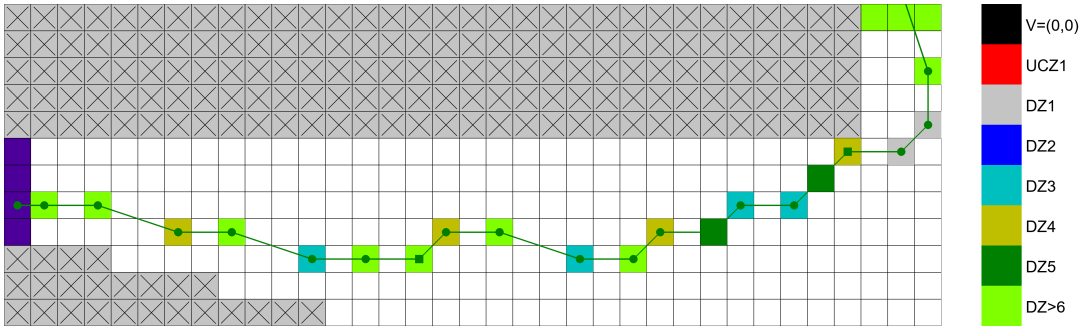
[40] Vivek Veeriah, Matteo Hessel, Zhongwen Xu, Janarthanan Rajendran, Richard L Lewis, Junhyuk Oh, Hado P van Hasselt, David Silver, and Satinder Singh. Discovery of useful questions as auxiliary tasks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Appendices

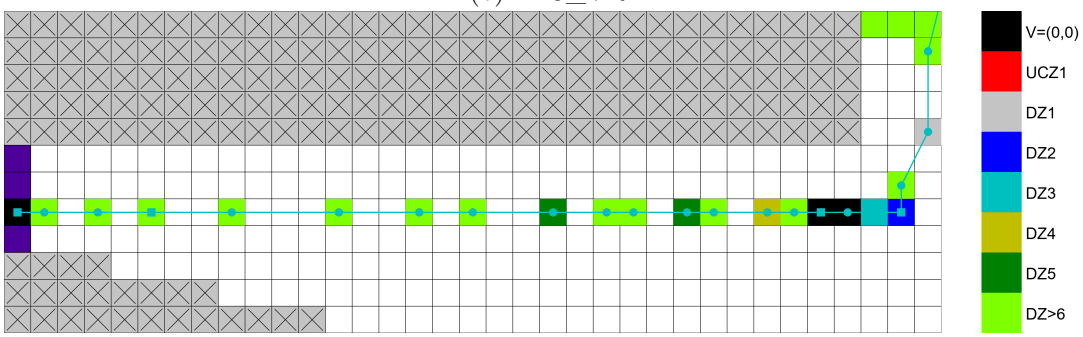
A. Path Illustrations



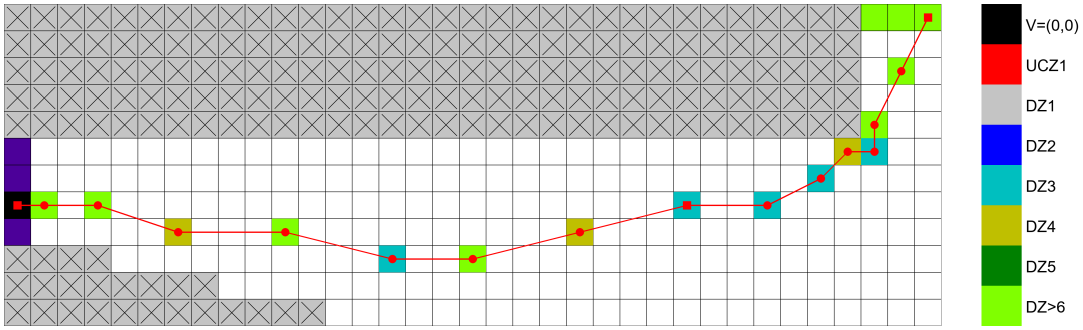
(a) A2C.



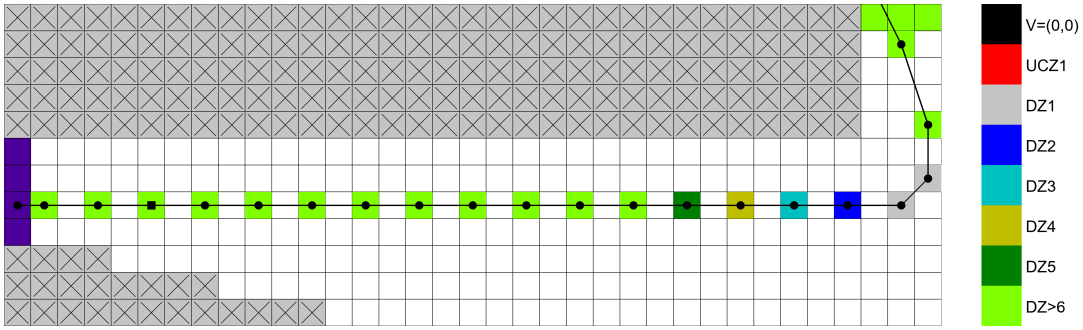
(b) A2C_VR.



(c) A2C_VR_DZ_UCZ.

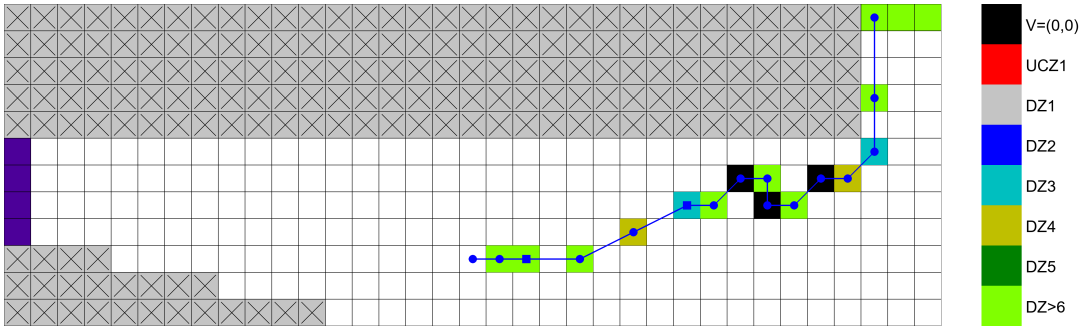


(d) DQN.

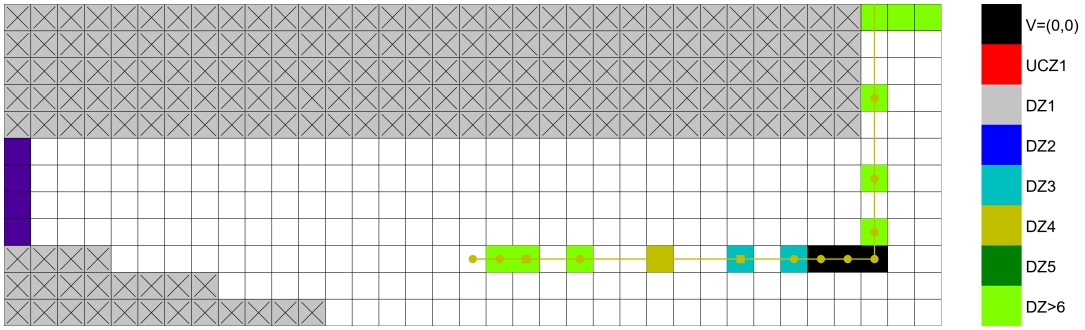


(e) UNREAL.

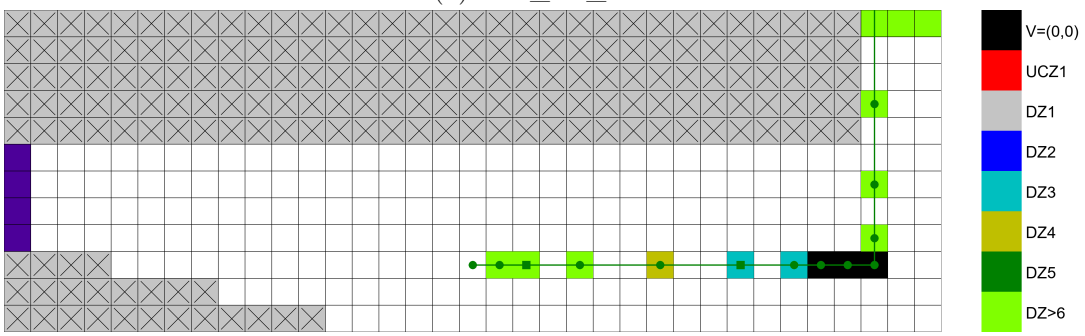
Figure A.1.: Exemplary DZ and UCZ paths in Barto-small map in NS N mode.



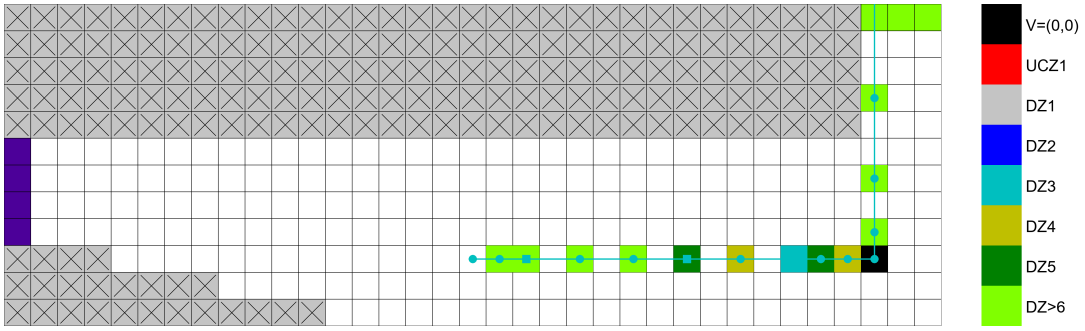
(a) A2C.



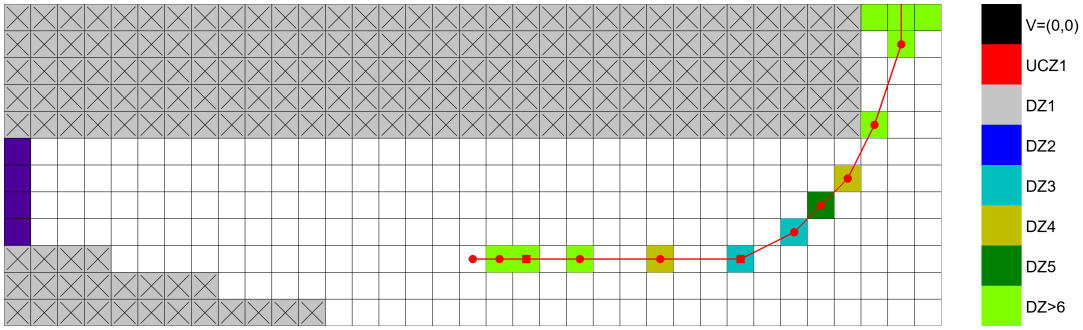
(b) A2C_DZ_UCZ.



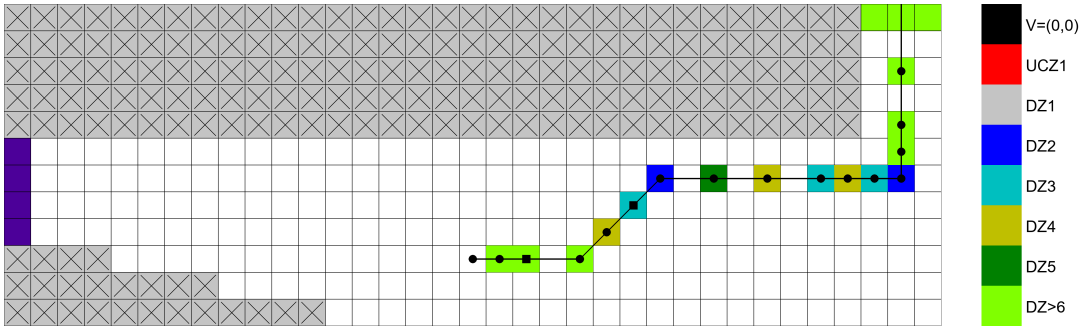
(c) A2C_VR.



(d) A2C_VR_DZ_UCZ.

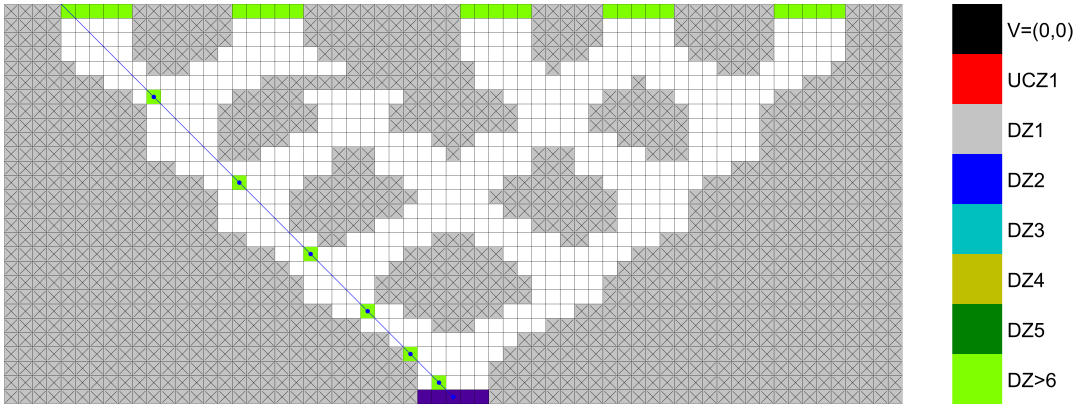


(e) DQN.

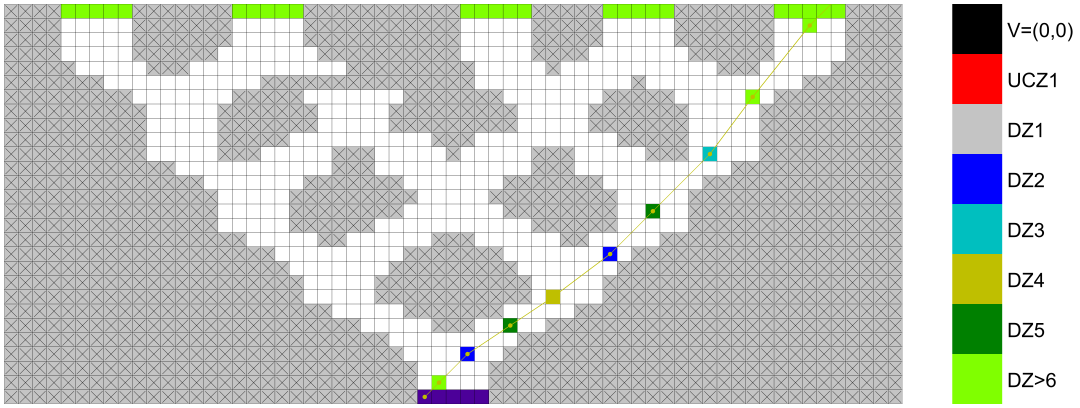


(f) UNREAL.

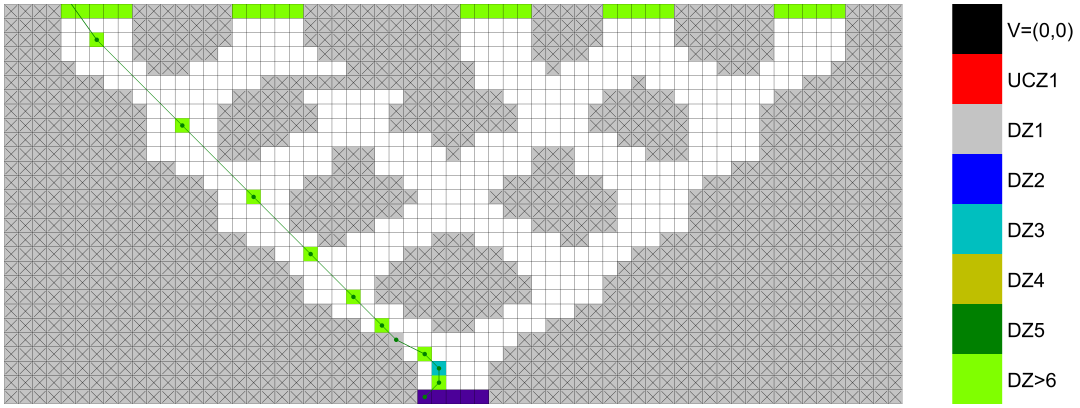
Figure A.2.: Exemplary DZ and UCZ paths in Barto-small map in RS N mode.



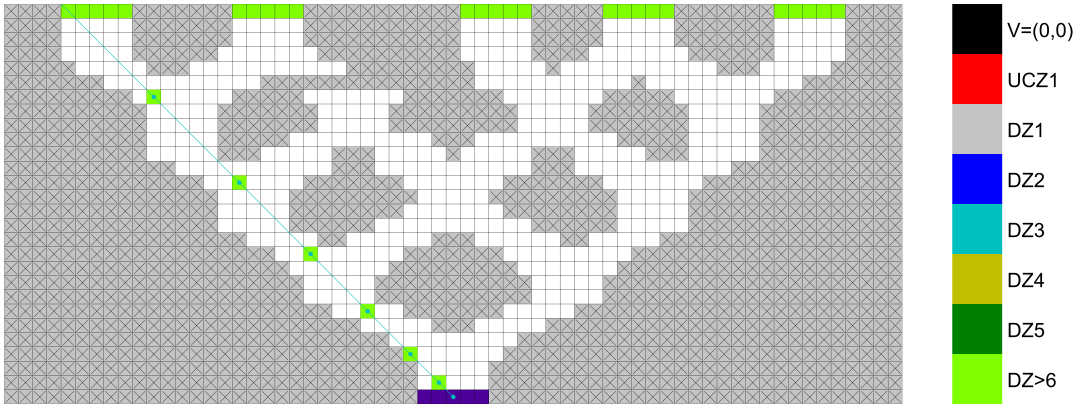
(a) A2C.



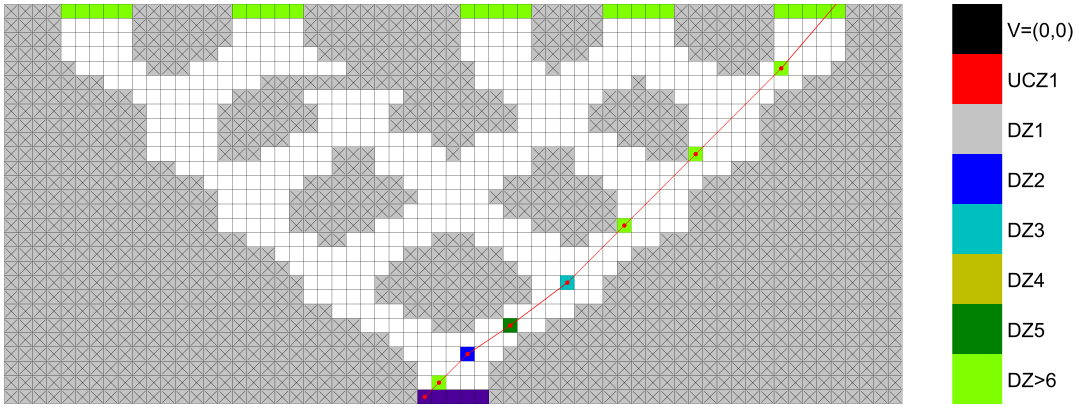
(b) A2C_DZ_UCZ.



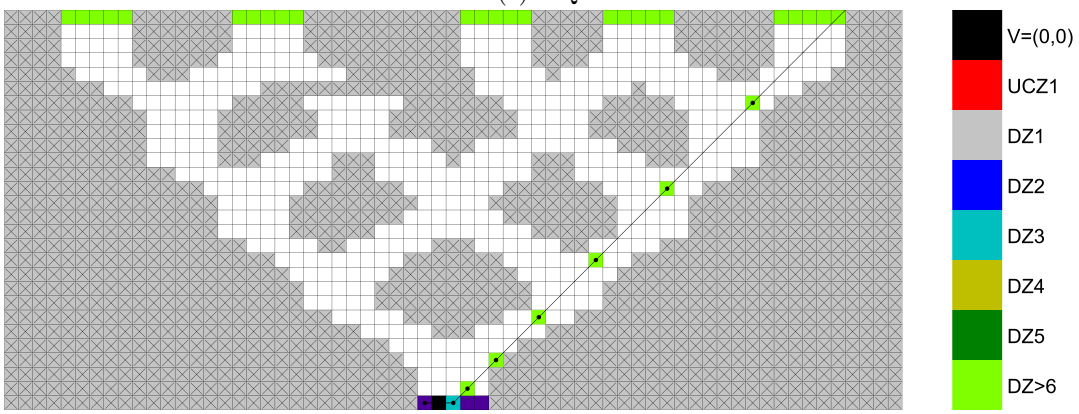
(c) A2C_VR.



(d) A2C_VR_DZ_UCZ.

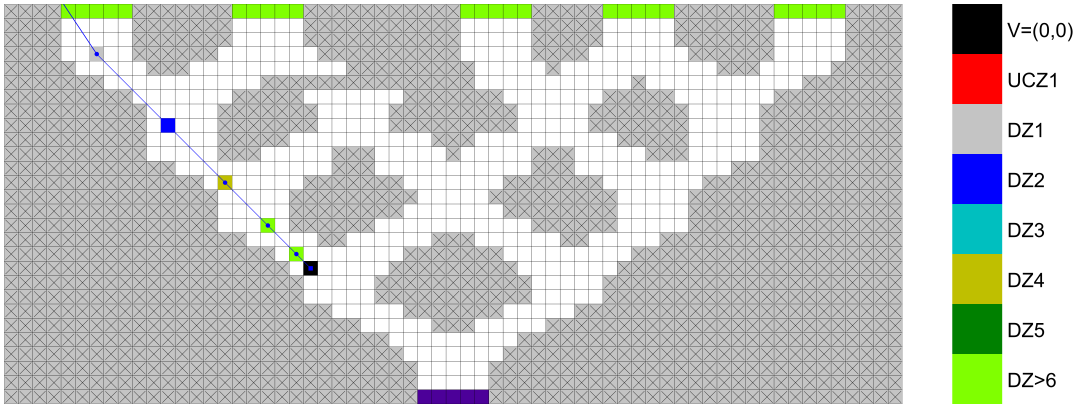


(e) DQN.

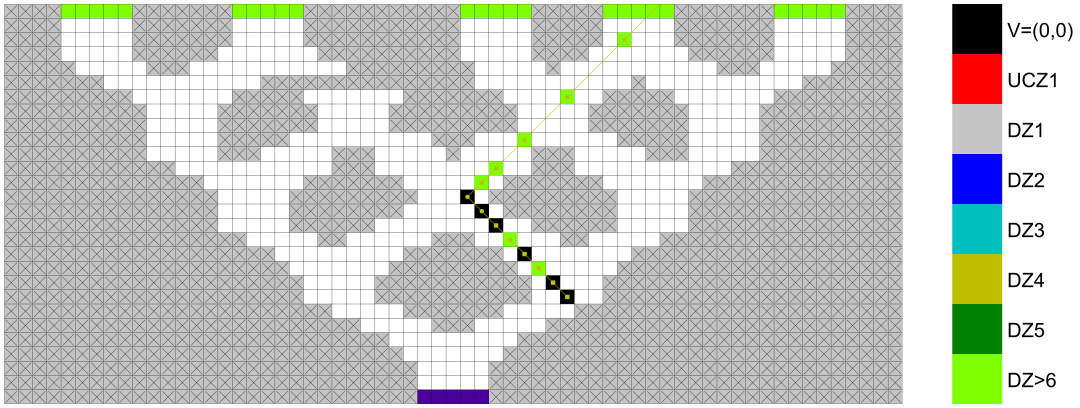


(f) UNREAL.

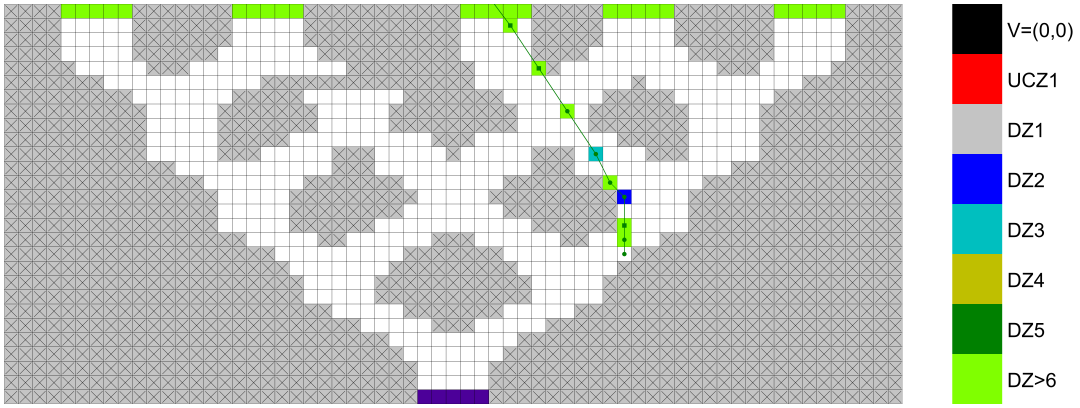
Figure A.3.: Exemplary DZ and UCZ paths in River-deadend map in NS N mode.



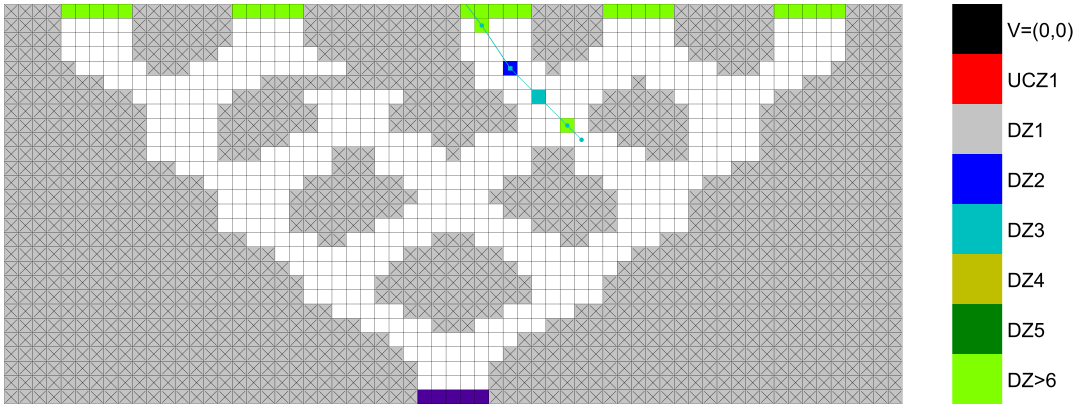
(a) A2C.



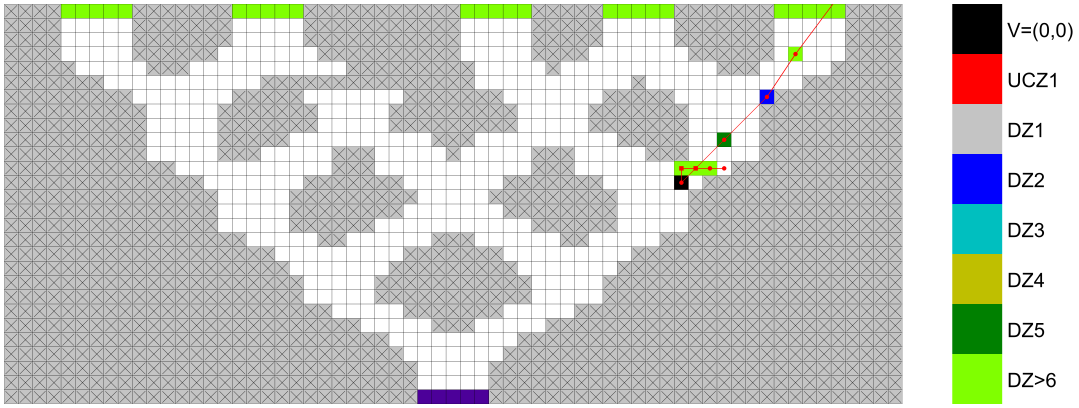
(b) A2C_DZ_UCZ.



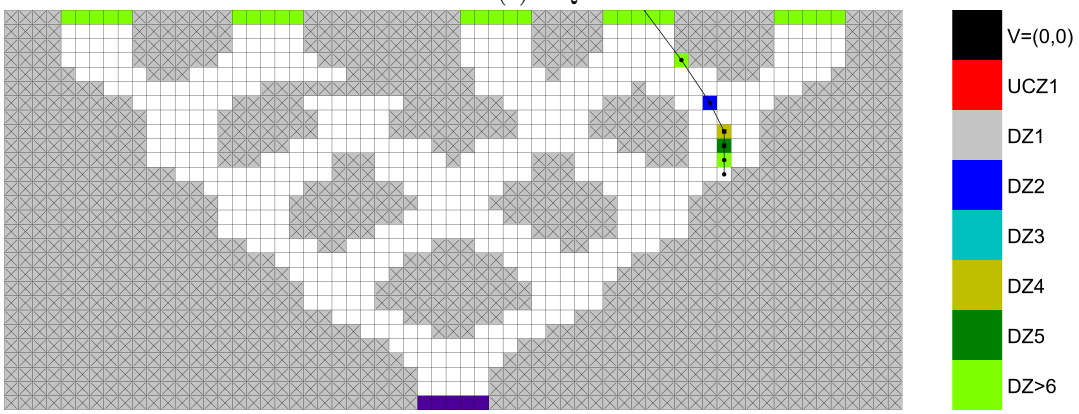
(c) A2C_VR.



(d) A2C_VR_DZ_UCZ.



(e) DQN.



(f) UNREAL.

Figure A.4.: Exemplary DZ and UCZ paths in River-deadend map in RS N mode.