Kristian Brudeli

Path-following and Collision Avoidance using World Models

Masteroppgave i Kybernetikk og robotikk
Veileder: Adil Rasheed
Medveileder: Thomas Nakken Larsen
Januar 2023

Kristian Brudeli

Path-following and Collision Avoidance using World Models

Masteroppgave i Kybernetikk og robotikk
Veileder: Adil Rasheed
Medveileder: Thomas Nakken Larsen
Januar 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

NTNU
Kunnskap for en bedre verden

Path-following and Collision Avoidance using World Models

Kristian Brudeli

January 2023

MASTER THESIS
Department of Engineering Cybernetics
Norwegian University of Science and Technology

Supervisor 1: Adil Rasheed

Supervisor 2: Thomas Nakken Larsen (co-supervisor)

Preface

This Master´s Thesis concludes a M.Sc. degree in Cybernetics and Robotics at NTNU Trondheim with a specialization in Navigation and Control of vehicles. The thesis is conducted without external funding.

This master thesis is not a direct continuation of the author's specialization project. The author's specialization project focused on a cloud-based pipeline Reinforcement Learning Research with a focus on reproducibility, scalability and collaboration.

The thesis builds upon the `gym-auv` software framework and the Master Thesis of Eivind Meyer [26], where publications such as [28], [27] and [21] are other contributions.

The reader of this work is expected to have knowledge at the level of a M.Sc. student or higher within Dynamical Systems and Differential Equations, probability theory as well as the fundamentals of Deep Learning. Knowledge of Reinforcement Learning and Stochastic Sequential Models is also beneficial, but an introduction to these preliminaries is also provided in the thesis.

Trondheim, 2023-01-27

Kristian Brudeli

Acknowledgment

I would like to acknowledge my main supervisor Adil Rasheed and my co-supervisor Thomas Nakken Larsen for their guidance and support during the writing of this master's thesis.

Further, I would like to acknowledge associate professor Albert Lau for allowing me to access powerful workstation GPU resources and Kenneth Sundli for his system administrator support on this workstation.

K.B.

Abstract

This master thesis presents an application of the model-based DreamerV2 deep reinforcement learning agent for path-following and collision avoidance for autonomous surface vessels. The vessel learns a world model describing the dynamics of itself and its surroundings, which it uses to learn the task of path-following and collision avoidance.

The learned neural network world model can also be used to predict the future sensor inputs of the vessel given a sequence of actions. This may allow for greater human interpretability, as the model may be used for visualizing predicted future sensor inputs, which is not an inherent capability of prior work using model-free reinforcement learning methods.

Experimentally testing our approach on a path-following and collision avoidance benchmark, our vessel has a success rate of 76.8%. We show the agent's capability to predict up to 45 seconds into the future, given a sequence of actions. The relative position of larger static obstacles and the path is typically predicted well, but the trained model struggles with smaller obstacles and moving vessels.

Sammendrag

Denne masteroppgaven presenterer en applikasjon av den modellbaserte forsterkningslæringsagenten DreamerV2 for banefølging og kollisjonsunngåelse med autonome overflatsfartøy. Fartøyet lærer gjennom interaksjon med omgivelsene en verdensmodell som predikerer dynamikken til seg selv og omgivelsene, som brukes til å lære å løse oppgaven.

Den lærte verdensmodellen kan også brukes til å rekonstruere fremtidige sensormålinger gitt en sekvens med pådrag. Dette kan gi økt menneskelig tolkbarhet, da modellen har mulighet til å forespeile fremtidige sensormålinger, noe tidligere tilnærminger som bruker modellfri forsterkningslæring ikke har mulighet til.

Ved evaluering av modellen finner vi at den etter å trenes klarer å løse oppgaven med en suksessrate på 76.8%. Vi viser modellens evne til å predikere fremtidige observasjoner i en tidshorisont på 45 sekunder. Posisjonen til båten i forhold til banen som skal følges og større statiske hindringer blir som regel predikert relativt godt, men den sliter med små statiske hindringer samt bevegelsen til andre skip.

Contents

List of Tables

Chapter 1

Introduction

Models of dynamical systems are the basic building blocks in the field of engineering cybernetics. Traditionally, hand-engineered models of systems form the basis of control systems by providing an understanding of the system that may be used to design rudimentary control systems such as PID controllers, while stricter performance requirements may lead to the engineering of more advanced controllers. Models of system dynamics also enable the creation of state estimators such as Kalman filters, which are widely used in control systems due to their ability to use the dynamics models to accurately estimate states.

In recent years, Deep Learning has solved problems previously deemed infeasible. From predicting the structure of proteins [20], generating images [31] [32] and videos [35] from text prompts. Large Language Models such as GPT-3 are able to learn the patterns of grammar, syntax, and tone of human language, solving tasks they were not explicitly trained to do through instruction [5].

Deep Reinforcement Learning uses Deep Learning to tackle problem settings where an agent interacts with a simulated or real-life environment, learning to solve tasks by maximizing the rewards it gets from the environment. Using this approach, one might do everything from solving continuous control benchmarks to more impressive feats such as beating humans at Go [34], real-time strategy games such as StarCraft 2 [40], Dota 2 [30] to realistic racing simulators such as Gran Turismo[12].

Further, creating safe, reliable, and affordable autonomous surface vessels is a topic of particular interest. The possibilities of such technology are significant. For instance, Yara claims that its autonomous container vessel Yara Birkeland replaces 40,000 truck trips every year, and reduces CO2 and NOx emissions [41]. Autonomous vessels may also be used in other settings, such as passenger ferries in urban environments [3] or for small vessels collecting data about the seabed[38].

1.1 Related work

Several efforts have been made at NTNU in recent years to solve such problems with Reinforcement Learning. Meyer et. al. provided an approach applying the model-free Proximal Policy Optimization (PPO) algorithm to solve the task of simultaneous Path-Following and Collision Avoidance in a stochastically generated environment consisting of both static and moving obstacles [26] [28]. Meyer et.al. also encoded the Convention on the International Regulations for Preventing Collisions at Sea into the reward function to reinforce behavior compliant with the convention [27]. Outside the domain of surface vessels, similar approaches have also been applied to Autonomous Underwater Vehicles (AUVs) [19] and to quadcopter drones [25].

Although we only consider Deep Reinforcement Learning approaches in this work, many algorithms to solve the problem of path-following and collision avoidance are available. We find [39] to give a comprehensive overview.

1.2 Motivation

Given the importance of models in traditional control theory and the capability of deep learning to learn complex relationships and tasks, we wish to find out what model-based Reinforcement Learning can accomplish.

While many of the Reinforcement Learning applications show impressive performance on the tasks they set out to solve, they are not satisfactory when it comes to their ability in showing their understanding of the situation. Using model-free reinforcement learning, they can provide the actions they intend to take and the expected value of these actions, but cannot tell what they expect the consequences of their actions to be. However, this is a feature of the RSSM model used in this work. An example of this is shown in fig. 1.1.

This thesis applies the model-based DreamerV2 Reinforcement Learning agent to the problem of path-following and collision avoidance. By interacting with an environment, it seeks to the dynamics and rewards of previously unseen environments. Notably, they are trained to be able to predict and reconstruct future observations in their environments. Through this capability, they can also communicate their intended behavior to humans, it provides an important step in the right direction for these methods to be able to eventually build trust among their users.

1.3 Outline

Our work has the following structure

- **Background:** We introduce the preliminaries of Reinforcement Learning and sequential stochastic models before introducing the Dreamer agent. We also present the model of the "Otter" Unmanned Surface Vehicle used in our experiments.
- **Method:** We present the Reinforcement Learning environment originally introduced in [26], and present our adaptions to the observation space and reward function. We also present a more structured approach to reward function tuning than what is presented in prior work, which we use to tune our reward function. Finally, we introduce the environment configuration and hyperparameters for the Dreamer agent used in the experiments.
- **Software pipeline:** We present our improvements to the software simulation framework, including a new renderer implementation used to make image observations. We summarize our significant efforts to increase the code quality of the framework. Additionally, we give a summary of our attempt to extend a non-official implementation of the Dreamer agent because it seemingly had desirable features.
- **Experiments and Results:** We show the final performance of our agent through metrics, behavior in episodes, and its ability to predict future observations given actions.
- **Discussion:** We discuss the design choices of our experiments, and point out our hypotheses of the causes of the failure modes of our model.
- **Summary, Conclusion, and Recommendations for Further Work:** We summarize our work and propose recommendations for further research in the field.

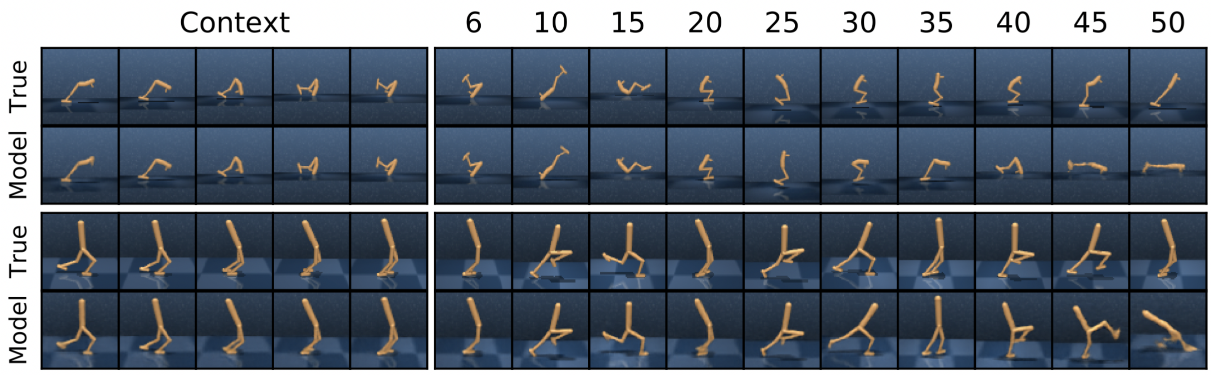


Figure 1.1: The Recurrent State Space Model used in Dreamer and PlaNet is able to generate accurate predictions of future observations in its environment. Given a short context of observations and the sequence of actions taken, a trained model can predict unseen future observations given the actions. This capability can both enhance trust in autonomous systems and enable timely intervention in situations where the predicted behavior is not satisfactory. Figure: [15]

Chapter 2

Background

2.1 Equations of Motion for Unmanned Surface Vessel

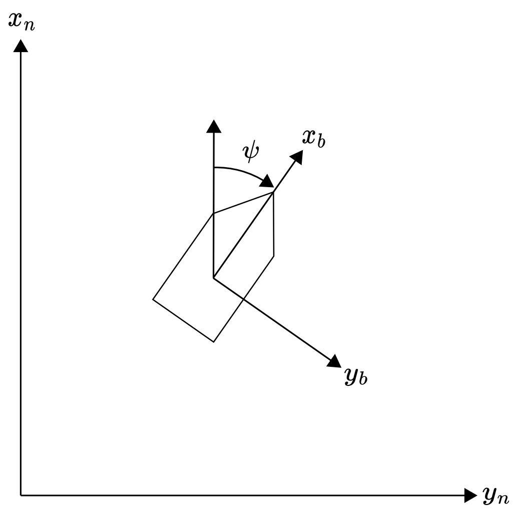


Figure 2.1: NED and Body coordinate frames.

This work uses a 3-DoF maneuvering model for the state and dynamics of the marine vessel. To simplify our experiments, we assume that the environment has

- No motion in roll, pitch or heave.
- No current or wind.
- Negligible actuator dynamics. The actuators apply the commanded forces without delay.
- No earth curvature or earth-induced coriolis effects.

As shown in [10], the dynamics of the vessel are governed by the equations

$$\dot{\eta} = \mathbf{R}(\psi)\nu \tag{2.1}$$

$$\mathbf{M}\dot{\nu} + \mathbf{C}(\nu)\nu + \mathbf{D}(\nu)\nu = \mathbf{Bf} \tag{2.2}$$

Here, $\eta = \begin{bmatrix} x^n & y^n & \psi \end{bmatrix}$ denote the coordinates in the NED frame and $\nu = \begin{bmatrix} u & v & \dot{\psi} \end{bmatrix}$, represent surge, sway and yaw rate. $\mathbf{M}$ is a coefficient matrix for mass, $\mathbf{C}(\nu)$ represents Coriolis effects, while $\mathbf{D}(\nu)$ represents dampening forces of water.

Where the mass matrix $\mathbf{M}$, the coriolis-centripetal acceleration matrix $\mathbf{C}$ and linear damping matrix $\mathbf{D}$ are given by

$$\mathbf{M} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & I_z \end{bmatrix} \qquad \mathbf{C}(\nu) = \begin{bmatrix} 0 & -mr & -mx_g r \\ mr & 0 & 0 \\ mx_g r & 0 & 0 \end{bmatrix} \tag{2.3}$$

$$\mathbf{D}(\nu) = \mathbf{D_L} + \mathbf{D_N}(\nu) \qquad \mathbf{D_L} = \begin{bmatrix} -X_u & 0 & 0 \\ 0 & -Y_v & 0 \\ 0 & 0 & -N_r \end{bmatrix} \qquad \mathbf{D_N}(\nu) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -10 \cdot N_r |r| \end{bmatrix} \tag{2.4}$$

We use an adapted version of the Otter model made by prof Thor I. Fossen [11]. The Otter Unmanned Surface Vehicle (USV) developed by Maritime Robotics, which features a propulsion system composed of two propellers, as may be seen in fig. 2.2. The forces exerted by the propellers are defined by the equation $\tau = \mathbf{Bf}$, where $\mathbf{B}$ represents the force configuration matrix and $\mathbf{f} = \begin{bmatrix} F_1, F_2 \end{bmatrix}^\mathrm{T}$ denotes the force vector. The matrix $\mathbf{B}$ effectively encodes that the propellers both produce forward force and that a moment induced by the difference in forces between the propellers. The force configuration matrix may be seen in eq. (2.5).

$$\mathbf{B} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ -l_1 & -l_2 \end{bmatrix} \tag{2.5}$$

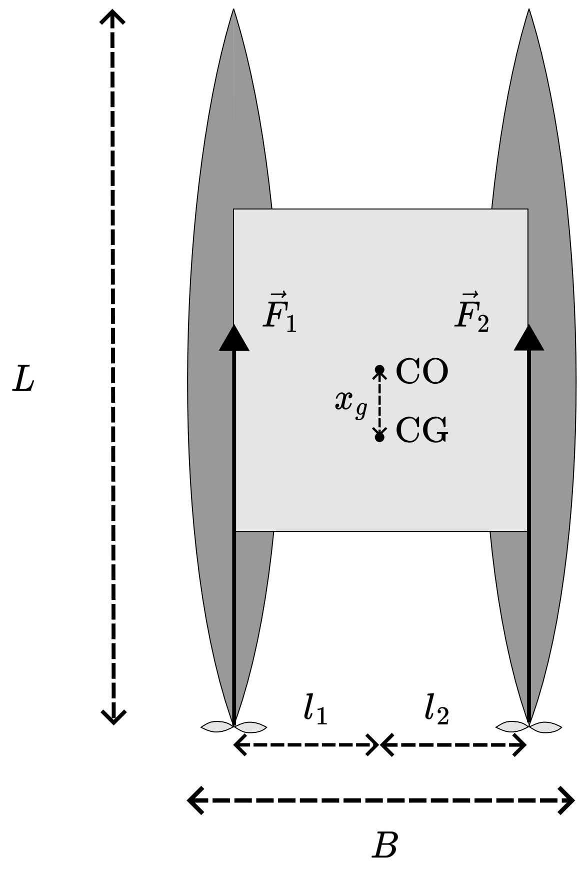


Figure 2.2: Illustration of the Otter USV

Symbol	**Value**	**Description**
m	55.0	Mass [kg]
I_z	13.75	Yaw-axis Moment of Inertia [kg $\cdot$ m^2]
x_g	0.2	Distance from CO to CG [m]
l_1	-0.395	Left thruster moment arm [m]
l_2	0.395	Right thruster moment arm [m]
L	2	Length [m]
B	1.08	Breadth [m]

Table 2.1: Otter model parameters

2.2 Sequential Stochastic Models

Predicting the future exactly and without uncertainty is typically not possible. Therefore, having models that can both predict and express the uncertainty of the future is desirable.

2.2.1 Markov assumption

The Markov Assumption eq. (2.6) is common for stochastic sequential models. Given exact knowledge of the current state-action pair (s_t, a_t), no information about prior time steps $s_{1:t-1}$ or actions $a_{1:t-1}$ can improve the knowledge of the distribution over the next state s_{t+1}. A model that fulfills the Markov assumption is said to be Markovian.

$$p(s_{t+1} \mid s_t, a_t) = p(s_{t+1} \mid s_{1:t}, a_{1:t}) \tag{2.6}$$

2.2.2 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) is a framework for modeling decision-making problems in which an agent must make decisions based on incomplete information. POMDPs modify the fully observable Markov Decision Process (MDP) by allowing for state variables that aren't observed directly, known as latent variables [36]. In other words, the agent's observations are not always sufficient to fully determine the system's current state. The model is Markovian, as we see in eq. (2.8).

$$\text{Prior:} \qquad p(s_0) \tag{2.7}$$

$$\text{Transition model:} \qquad p(s_{t+1} \mid s_t, a_t) \tag{2.8}$$

$$\text{Emission model:} \qquad p(o_t \mid s_t) \tag{2.9}$$

$$\text{Reward model:} \qquad p(r_{t+1} \mid s_t, a_t) \tag{2.10}$$

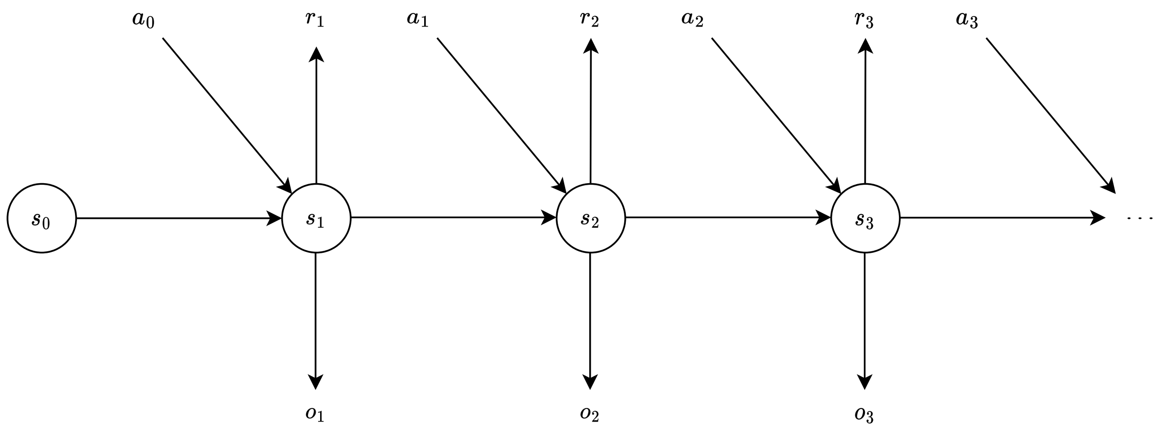


Figure 2.3: The structure of a Partially Observable Markov Decision Process (POMDP). The state s_t is influenced by the previous state s_{t-1} and the previous action a_{t-1}. Similarly, the reward depends also depends on the previous state and action. The observation o_t only depends on the corresponding state s_t.

2.3 Reinforcement Learning

Reinforcement Learning deals with learning how to act upon an environment to maximize expected rewards now and in the future. We denote $s \in S$ as the states, $a \in A$ as the actions, and $R : S \times A \times S \rightarrow \mathcal{R}$ as the reward function, with $r_t = R(s_t, a_t, s_{t+1})$[1]. The stochastic policy π may be used to sample actions based on the current state, i.e., $a_t \sim \pi(\cdot \mid s_t)$. We assume that the environment's reward function and dynamics are unknown to the agent. [1]

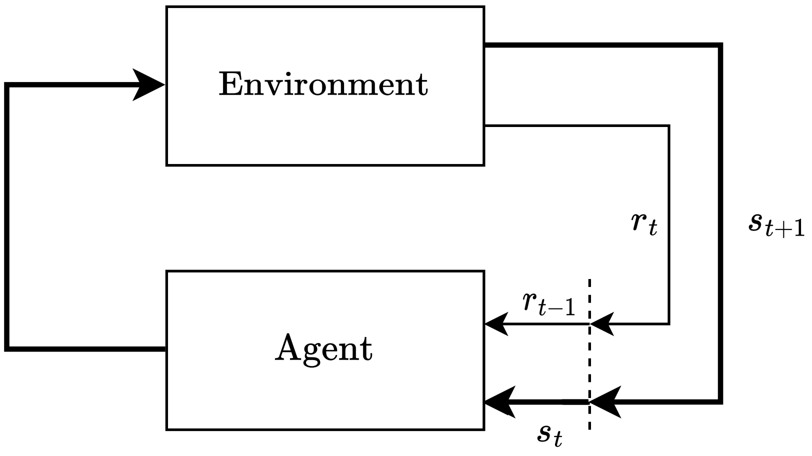


Figure 2.4: Loop showing the standard reinforcement learning setup. An agent takes actions a and the environment returns states s and rewards r.

To formulate Reinforcement Learning as a clearly defined optimization problem, one must decide how to summarize the current and future rewards as a single number. This scalar is referred to as the return $R(\tau)$. The finite-horizon undiscounted return is denoted by eq. (2.11), while the infinite-horizon discounted return is denoted by eq. (2.12) [1].

Finite horizon:
$$R(\tau) = \sum_{t=0}^{T} r_t \tag{2.11}$$

Inifinite horizon, discounted:
$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \quad 0 \leq \gamma < 1 \tag{2.12}$$

[1]The seminal work by [36] defines the reward function as $(s_{t+1}, r_{t+1}) \sim p(s_{t+1}, r_{t+1} \mid s_t, a_t)$ to highlight the interdependence between the state transition $s_t \rightarrow s_{t+1}$ and the reward received. In this work, we seek to keep the equations and implementations as similar as possible to avoid confusion. Therefore, we stick to the definition used in the API of the commonly used gym library by research institute OpenAI, i.e. $(s_{t+1}, r_t) \sim p(s_{t+1} r_t \mid s_t, a_t)$. [4]

2.3.1 Value and Advantage functions

Value functions denote the expected return in the current state. There are multiple variations of value functions. The on-policy value function $V^{\pi}(s)$ eq. (2.13) gives the expected return if we start in state s and apply actions from the policy π. The on-policy action-value function $Q^{\pi}(s,a)$ eq. (2.14) is similar and gives the expected return if the action a is done in the current state s while using actions from the policy π in future time steps. The optimal value function $V^{*}(s)$ eq. (2.15) gives the expected return of an agent that starts in state s and follows an optimal policy π^{*}, i.e. a policy that maximizes the expected reward. Similarly, the optimal action-value function $Q^{*}(s,a)$ eq. (2.16) denotes the expected reward if an arbitrary action a is taken in state s, while optimal actions are taken thereafter.

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau \mid s_0 = s)\right] \tag{2.13}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}\left[R(\tau \mid s_0 = s, a_0 = a)\right] \tag{2.14}$$

$$V^{*}(s) = \max_{\pi}\ \mathbb{E}_{\tau\sim\pi}\left[R(\tau \mid s_0 = s)\right] \tag{2.15}$$

$$Q^{*}(s,a) = \max_{\pi}\ \mathbb{E}_{\tau\sim\pi}\left[R(\tau \mid s_0 = s, a_0 = a)\right] \tag{2.16}$$

From this, we also see that these are related, for instance, we have that

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}\left[Q(s,a)\right] \tag{2.17}$$

$$V^{*}(s) = \arg\max_{a}\left[Q^{*}(s,a)\right] \tag{2.18}$$

The advantage function $A^{\pi}(s,a)$ describes the difference of the expected return of taking the action a in state s compared to the expected return of sampling an action from the policy π, given that subsequent actions are sampled from same policy π. In other words, the advantage function lets us assess the actions while taking into account that some states already have a higher expected return, essentially isolating the effect of the action on the return. [1]

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.19}$$

2.3.2 Model-free methods

Model-free methods approach the reinforcement learning problem by trying to learn a mapping from states to action directly, and optimize in order to maximize the expected rewards without learning the dynamics.

Value-based methods

Value-based methods approximate value functions, and uses them to pick actions a. In principle, if we knew the optimal value-action function $Q^*(s,a)$, it could then be used as an optimal policy. This would be done by using the value function to find actions $a^* = \arg\max_a Q^*(s,a)$ for the current state s at every time step and apply them to the environment. While this may work well with discrete action spaces, it may often be challenging to do in large, continuous action spaces.

Policy optimization

Policy optimization tries to optimize the policy π directly. If we sample actions $a_t \sim \pi_\theta(s_t)$ from a parameterized policy π_θ, we may use stochastic gradient descent to optimize θ. If we have a stochastic policy π_θ with a loss function of its expected return $J(\pi) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t]$, it can be shown to have the gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[R_t \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{2.20}$$

Actor-critic methods

Actor-critic methods combine ideas from value-based methods and policy optimization. For instance, by learning both a policy π_θ and an on-policy value function $V^\pi(s)$, one may calculate an advantage estimate $\hat{A}^{\pi_\theta}$ to find out how good the return of the action was compared to the expected return of the policy. In other words, by reinforcing actions that perform better than expected, we iteratively improve the policy's performance.

2.3.3 Model-based methods

Model-based Reinforcement Learning methods learn a model of the environment to solve the Reinforcement Learning problem. Although model-free methods such as Proximal Policy Optimization (PPO) [33] often learn mappings from states s_t to actions a_t to perform tasks quite well, they do not learn the dynamics of the environment, e.g. they cannot predict state transitions such as $s_{t+1} = f(s_t, a_t)$.

The dynamics of an environment are conditionally independent of the policy given the actions. This means that if we take an action a_t in the state s_t, the transition $s_{t+1} \sim f(s_t, a_t)$ does not depend on *how* the action a_t was chosen. Similar arguments also apply to the reward. Because of this, model-based methods may often also work with off-policy data, as the estimates of the dynamics and rewards of the environment may be improved through rollouts collected using other policies. We note that some model-free agents, like DQN [29] also can learn from off-policy data, but only learns value models and not dynamics models.

2.3.4 Exploration vs. Exploitation

An important problem that must be tackled in reinforcement learning problems is the exploration vs. exploitation trade-off. As the agent does not have any prior knowledge of its environment, it must learn how to optimally operate in an environment by interacting with it. In this context, exploitation refers to taking actions that maximize the current expected return according to the policy. However, it is unlikely that the optimal behavior is immediately apparent for the policy π to exploit. This calls for exploration, i.e. intentionally deviating from what is expected to be the optimal behavior to increase the chance of getting unintended behavior not suggested by our current policy.

2.4 Dreamer

DreamerV2 is a model-based Reinforcement Learning algorithm that learns a world model from experiences, which is then used to learn how to act in the environment through imagined experiences. In other words, instead of learning behaviors through real interactions with the environment, its behaviors are learned through training on a neural network world model. This section explains the most important aspects of the DreamerV2 agent. We aim to point out some important details that are covered in the original papers but may be overlooked by someone who is new to stochastic sequential models. Still, we refer readers to the original paper [16] as well as the papers leading up to it [15] [14] for a more thorough introduction.

2.4.1 Modes of operation

Dreamer has several modes of operation, which are repeated during training. Dreamer typically collects some specified amount of experience (such as some number of time steps or a complete episode) using environment rollouts before it trains for some predefined time using world-model learning and actor-critic learning. It then repeats the cycle, alternating between collecting more experiences and training.

- **Environment rollouts** The agent collects experiences from the environment to use for world-model learning later. See fig. 2.5.
- **World model learning** The RSSM world model is trained from previous experiences without interaction with the environment. See fig. 2.6a.
- **Actor-critic learning** The agent uses the RSSM world model to simulate *imagined* trajectories that only take place inside the world model, not the actual environment. These imagined trajectories are used to learn behavior through actor-critic learning. See fig. 2.6b.

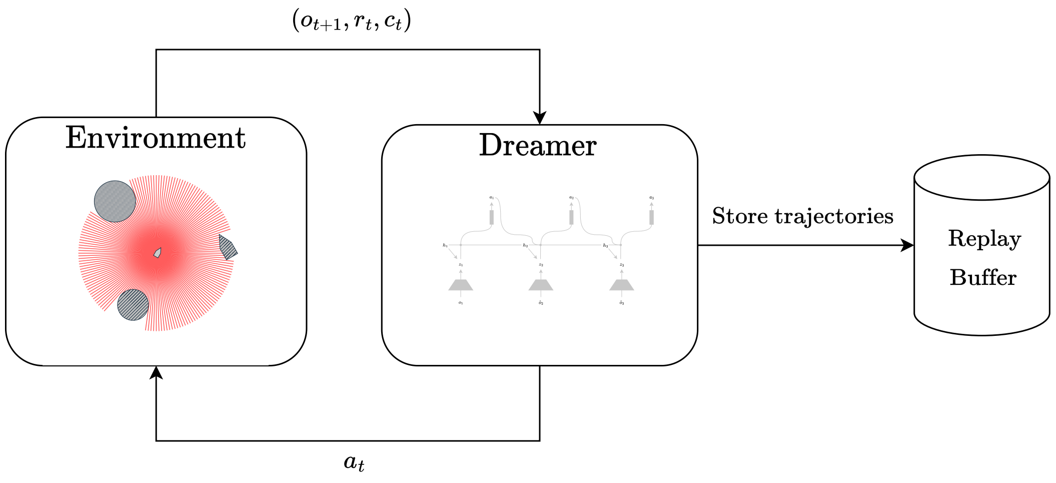
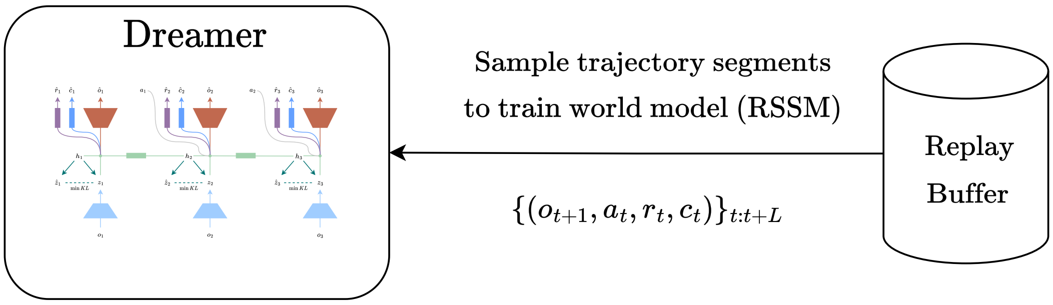
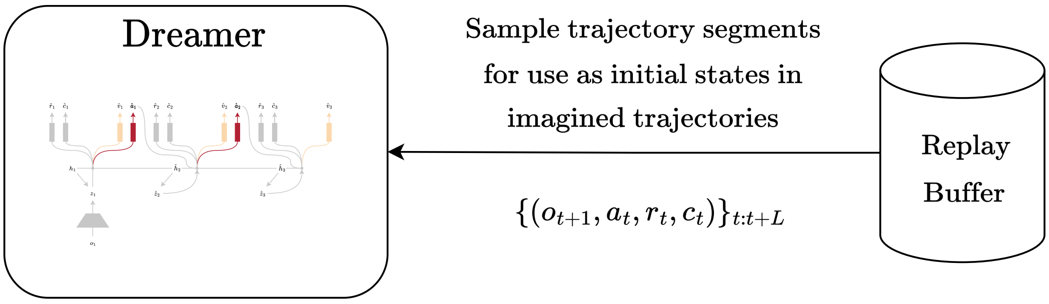


Figure 2.5: Environment rollout mode.


Dreamer
Sample trajectory segments
to train world model (RSSM)
$\{(o_{t+1}, a_t, r_t, c_t)\}_{t:t+L}$
Replay
Buffer

(a) World-model learning mode.


Dreamer
Sample trajectory segments
for use as initial states in
imagined trajectories
$\{(o_{t+1}, a_t, r_t, c_t)\}_{t:t+L}$
Replay
Buffer

(b) Actor-critic learning mode.

Figure 2.6: Dreamer modes of operation.

2.4.2 Recurrent State-Space Model

Dreamer uses a Recurrent State-Space Model (RSSM) as its world model. The RSSM was originally introduced in [15], which used it together with MPC-style planning to solve control tasks. Dreamer uses this model to learn an actor-critic policy through imagined trajectories.

$$
\begin{aligned}
&\text{Recurrent model:} && h_t = f_\phi(h_{t-1}, z_{t-1}, a_{t-1}) && (2.21)\\
&\text{Representation model:} && z_t \sim q_\phi(z_t \mid h_{t-1}, o_t) && (2.22)\\
&\text{Transition predictor:} && \hat{z}_t \sim p_\phi(\hat{z}_t \mid h_t) && (2.23)\\
&\text{Observation predictor:} && \hat{o}_t \sim p_\phi(\hat{o}_t \mid h_t, z_t) && (2.24)\\
&\text{Reward predictor:} && \hat{r}_t \sim p_\phi(r_t \mid h_t, z_t) && (2.25)\\
&\text{Continue predictor:} && \hat{c}_t \sim p_\phi(c_t \mid h_t, z_t) && (2.26)
\end{aligned}
$$

The Recurrent State Space Model (RSSM) models its environments similarly to a POMDP. We give a summary of its components.

- **Recurrent model:** The recurrent model predicts the deterministic part of the transition model eq. (2.8). This deterministic part of the state is denoted h_t. It uses a Gated Recurrent Unit (GRU), a type of Recurrent Neural Network.[7]
- **Representation model:** The representation model uses the deterministic part of the previous hidden latent state h_t and the current observation o_t to represent z_t, the stochastic part of the latent state. z_t is also referred to as the *posterior*.
- **Transition model:** If the observation o_t is not available to calculate the stochastic latent state z_t, we make $\hat{z}_t$, an approximated prediction of the stochastic part of the latent state. $\hat{z}_t$ is also referred to as the *prior*.
- **Observation predictor:** The observation predictor uses the latent state (h_t, z_t) to predict/reconstruct the observation $\hat{o}_t$. The observation is typically an image or a feature vector. This is similar to the emission model in a POMDP eq. (2.9).
- **Reward predictor:** Similarly, the reward predictor uses the latent state (h_t, z_t) to predict the reward $\hat{r}_t$.
- **Continue predictor:** The trajectories τ are not of fixed length and may end at any of the time steps. This may, for instance, be due to the agent being destroyed or if it completes the task. The continue predictor predicts whether this will happen at some time step. We will explain the use of this later in section 2.4.7.

The latent state of the model consists of a deterministic part h_t and a stochastic part z_t. The combination of these two parts allows the model to predict trajectories where the dynamics are primarily deterministic but may also have some uncertainty or process noise. The deterministic part of the latent state, h_t, is a fixed-size vector. z_t, the stochastic state, is a vector of reparameterized categorical distributions.

2.4.3 Environment rollouts

The algorithm collects data for its replay buffer from running the policy with additive exploration noise. However, it is worth noting that although Dreamer collects data for the replay buffer using the policy, the policy is trained using imagined trajectories from the world model, which we will explain later in section 2.4.6. This differs from conventional Deep Reinforcement Learning algorithms such as PPO, SAC, or DQN that directly use environment interactions (either online or offline by using a replay buffer) to learn their value functions and policies.

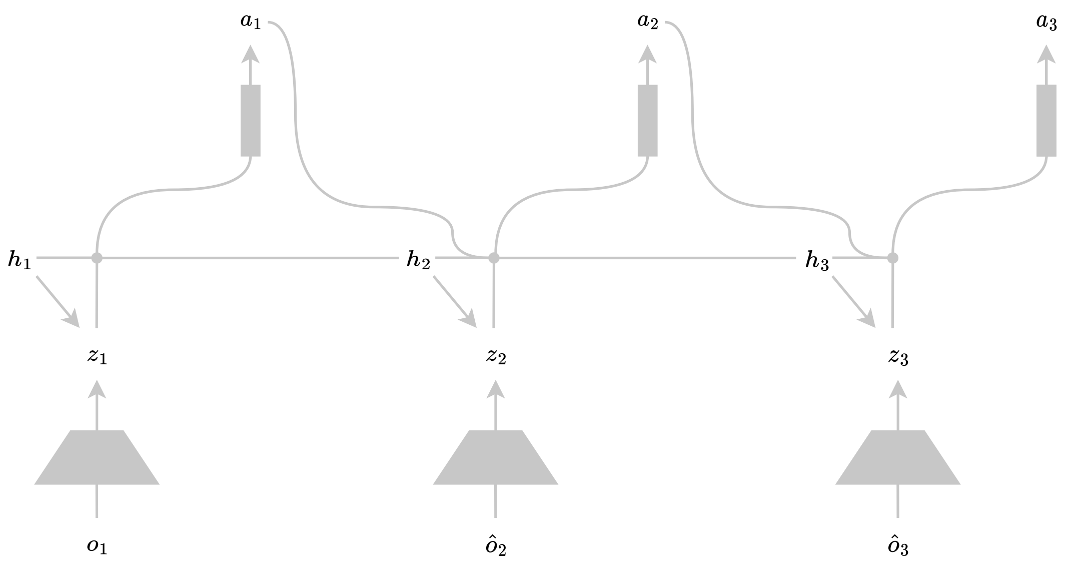


Figure 2.7: Components used by Dreamer during environment rollouts. Environment rollouts are the only time Dreamer interacts with the environment. No parameters are trained during environment rollouts. Tuples of actions, observations, rewards, and continuation labels (a, o, r, c) are collected and added to a replay buffer for world-model learning later.

2.4.4 Parameter sets

Before going into the different stages of how Dreamer is trained, we pause to point out how each stage optimizes only over one set of parameters. To highlight this, we color the parts of the model trained during a stage. Parts of the model that are used but not optimized over are colored gray. We point out that we might still calculate the gradients through the frozen (gray) parameters to optimize other (colored) parameters. This is most notably done in fig. 2.9, where we differentiate through the world-model parameters ϕ to optimize the actor parameters ψ.

Symbol	**Name**	**Colors in illustrations when trained**
ϕ	RSSM	🟦 🟩 🟩 🟧 🟪 🟦
ψ	Actor	🟥
ξ	Critic	🟧

Table 2.2: Parameter sets

2.4.5 World Model Learning

In the general case, the world model in Dreamer is trained according to the loss function

$$\mathcal{L}(\phi) = E_{q_\phi(z_{1:T} | o_{1:T}, a_{1:T})}\Bigg[\sum_{t=1}^{T} \underbrace{-\ln p_\phi(o_t \mid h_t, z_t)}_{\text{Observation log loss}} \underbrace{-\ln p_\phi(r_t \mid h_t, z_t)}_{\text{Reward log loss}} \underbrace{-\ln p_\phi(c_t \mid h_t, z_t)}_{\text{Continue log loss}}$$
$$\underbrace{+\beta KL\big[q_\phi(z_t \mid h_t, o_t) \,||\, p_\phi(z_t \mid h_t)\big]}_{\text{Prior/posterior KL divergence loss}}\Bigg] \tag{2.27}$$

This formulation allows for a wide range of spaces to be modeled by the loss, e.g., continuous or discrete observations. We consider a special case that will be relevant later, where the observations and rewards are continuous and modeled using univariate Gaussian distributions. In this case, the loss reduces to eq. (2.28).

$$\mathcal{L}(\phi) = E_{q_\phi(z_{1:T} | o_{1:T}, a_{1:T})}\Bigg[\sum_{t=1}^{T} \underbrace{||o_t - \hat{o}_t||^2}_{\text{Observation log loss}} + \underbrace{||r_t - \hat{r}_t||^2}_{\text{Reward log loss}}$$
$$\underbrace{-c_t \ln(\hat{c}_t) + (1 - c_t) \ln(1 - \hat{c}_t)}_{\text{Continue log loss}}$$
$$\underbrace{+\beta KL\big[q_\phi(z_t \mid h_t, o_t) \,||\, p_\phi(z_t \mid h_t)}_{\text{Prior/posterior KL divergence loss}} + \text{const}\big]\Bigg] \tag{2.28}$$

In eq. (2.28) $c_t \in \{0, 1\}$ are the true continue labels and $\hat{c}_t \in (0, 1)$ are the estimated probabilities that the corresponding episodes end at time step t.

We see that the loss function reduces to the mean-squared error between the actual values and predictions for the continuous observations and rewards, while the binary continue predictor uses the mean binary cross-entropy loss. A constant is also present in the loss but is irrelevant during optimization.

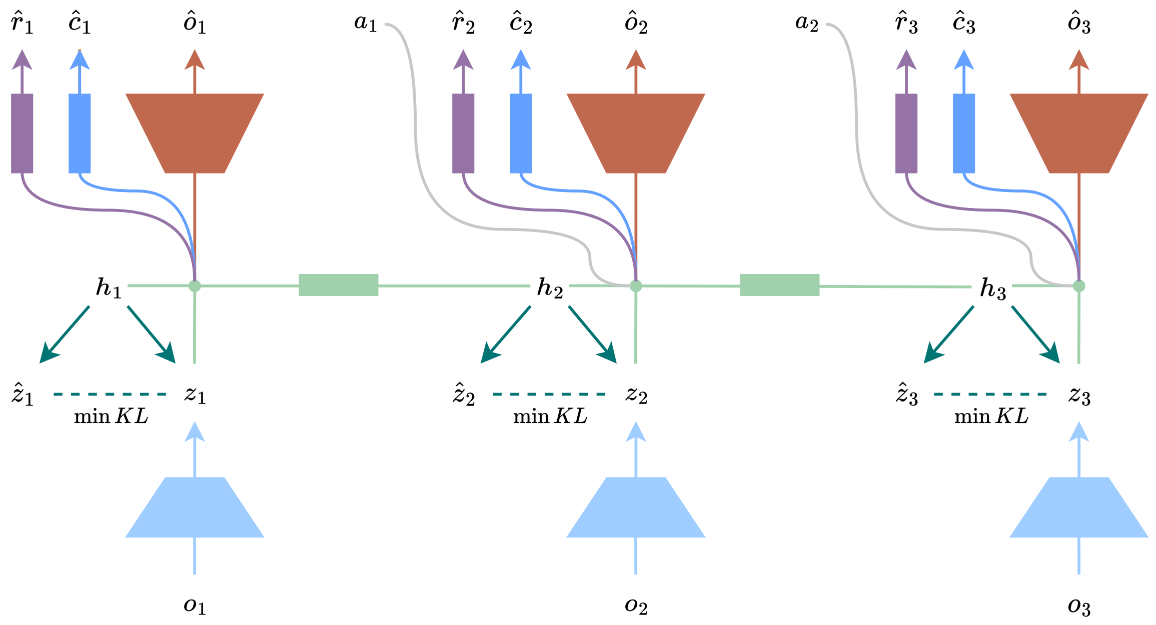


Figure 2.8: World model learning. Trajectories are sampled from the replay buffer, and the model jointly learns the world model through eq. (2.27). Predictions of rewards $\hat{r}$, continue predictions $\hat{c}$ and observations $\hat{o}$ are made and trained towards their true values. Latent state priors $\hat{z}_t \sim p(\hat{z}_t \mid \hat{h}_t)$ and posteriors $z_t \sim p(z_t \mid h_t, o_t)$ are kept similar through minimizing KL-divergence. This regularizes the dynamics model, keeping the transition and representation models similar.

2.4.6 Actor-Critic Learning

Dreamer learns behavior purely through imagined trajectories predicted by its world model without reconstruction. A visualization of the involved components may be seen in fig. 2.9. The approach samples observations from the replay buffer as initial states, encoded into latent states z_t using the representation model eq. (2.22). The rest of the trajectories are predicted using the transition predictor eq. (2.23), with actions $\hat{a}_t$ sampled from the policy π_ψ.

The actor-critic approach of Dreamer has multiple benefits compared to conventional policy learning, such as sample efficiency and parallelizability.

In model-free RL methods, the only way to collect more experience is through more interactions with the environment. In model-based RL, we may use the world model to simulate or "imagine" experiences in an environment, thus learning behaviors without actually interacting with the environment, increasing sample efficiency.

As we use a fixed-size model in the RSSM and implement it using tensor operations, it is massively parallelizable using accelerators such as GPUs or TPUs. We imagine trajectories from each of the $B \cdot L$ time steps sampled from the replay buffer. With the hyperparameters in the original paper, $B \cdot L = 2500$ trajectories are imagined in parallel on a single GPU for $H = 15$ time steps each, such that 37500 time steps are imagined across trajectories each iteration. Although the model *can* reconstruct imagined trajectories, this is not done during actor-critic learning. Only the predicted latent states $(\hat{h}_{1:H}, \hat{z}_{1:H})$ are needed to predict the rewards $\hat{r}_{1:H}$, values $\hat{V}_{1:H}$ and discount rate predictions $\hat{\gamma}_{1:H}$ used for learning the policy π_ψ.

The value targets in DreamerV2 are V^λ-targets [36], but with predicted discounts. [2]

$$V_t^\lambda = \begin{cases} \hat{r}_t + \hat{\gamma}_t((1-\lambda)v_\xi(\hat{z}_H)) + \lambda V_{t+1}^\lambda & \text{if} \quad t < H \\ \hat{\gamma}_t v_\xi(\hat{z}_H) & \text{if} \quad t = H \end{cases} \tag{2.29}$$

The critic loss is given by

$$\mathcal{L}(\psi) = E_{p_\phi, p_\psi}\left[\sum_{\tau=1}^{H-1} \frac{1}{2} \|v_\xi(s_\tau) - \operatorname{sg}(V_\lambda(s_\tau))\|\right] \tag{2.30}$$

Here, sg is the "stop gradient"-operator. The actor loss differentiates through the dynamics, as the value for each time step is a weighted sum of rewards and values as seen in section 2.4.6.

[2] Our equation corrects a typo from the original paper where the reward was added to the bootstrapped value for V_H^λ. The original author discusses the typo and confirms the correction in a discussion related to the official implementation of DreamerV2 on GitHub. [13]

Furthermore, the actor loss is given by

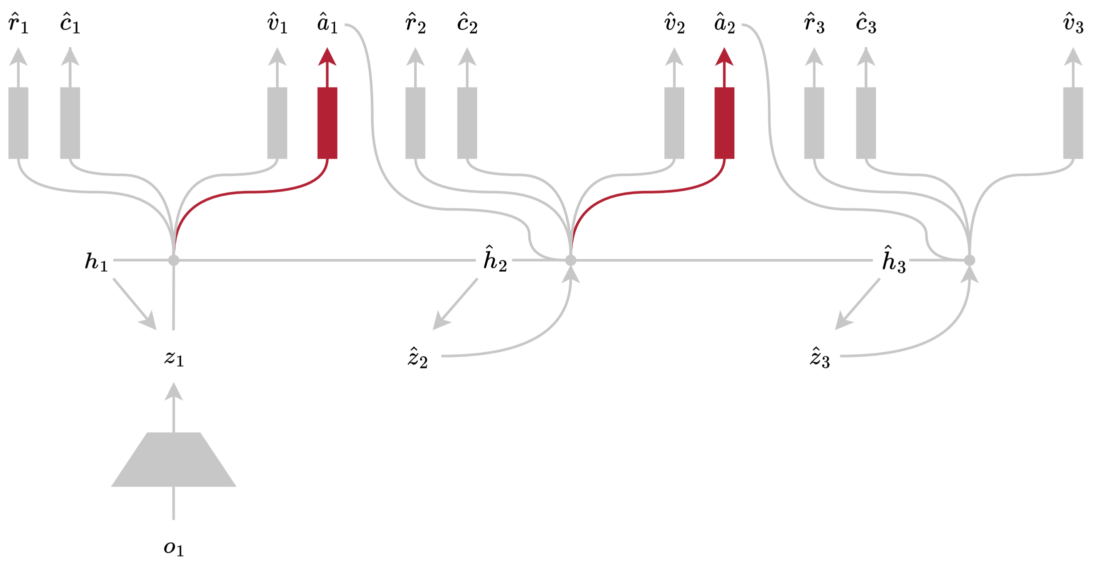
$$\mathcal{L}(\xi) = E_{p_\phi, p_\psi}\left[\sum_{t=1}^{H-1} \underbrace{\left(-\rho \ln p_\psi(\hat{a}_t \mid \hat{z}_t)\mathrm{sg}(V_t^\lambda - v_\xi(\hat{z}_t)\right)}_{\text{Reinforce}} \underbrace{-V_\lambda(z_\tau)}_{\text{Dynamics backprop}} \underbrace{-\eta H\left[a_t \mid \hat{z}_t\right]}_{\text{Entropy bonus}}\right] \tag{2.31}$$

2.4.7 Continuation prediction

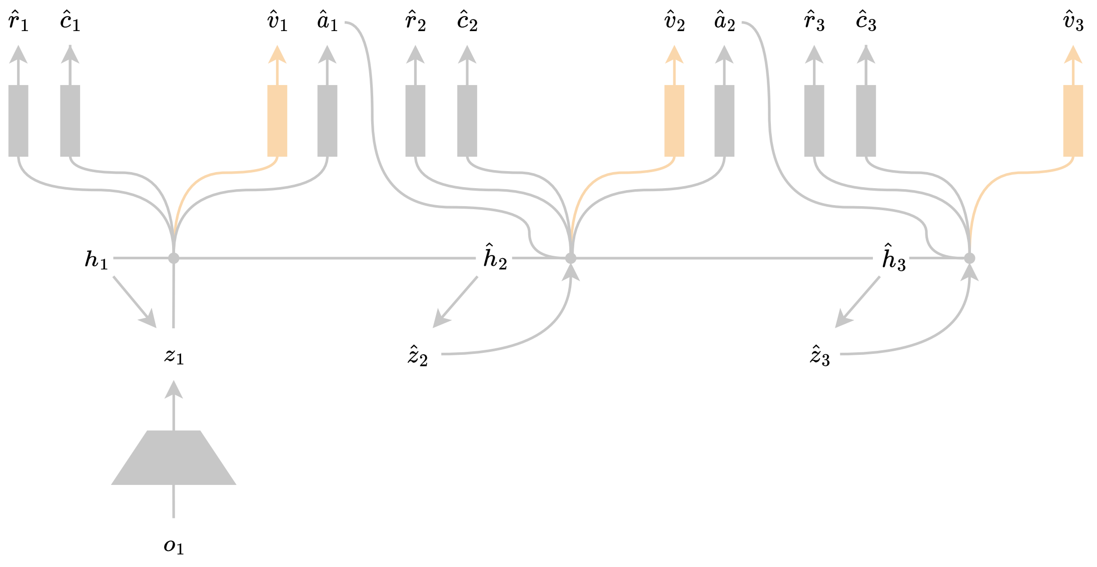
Dreamer imagines trajectories of a fixed-length horizon H. To account for the possibility that the episode ends, the world model predicts continuation[3] labels $c_{1:H}$. We shift the predicted continuation labels to the right and drop the last one, as a label $c_t = 0$ indicates that the *next* predicted discount rate $\hat{\gamma}$ should be zero. As the initial state of each imagined trajectory is sampled from the replay buffer, as shown in fig. 2.6b, we use the actual continue label c_1 for the first time step. We use these to construct predicted discount rates $\hat{\gamma}_{1:H}$ according to

$$\hat{\gamma}_t = \gamma\left(1 - E_{p_\phi, p_\psi}\left[\hat{c}_{t-1} \mid \hat{h}_{t-1}, \hat{z}_{t-1}\right]\right) \tag{2.32}$$

[3]The original DreamerV2 paper refers to this as discount prediction. In practice, it is implemented as a binary predictor for if the episode ends, which motivates our choice to call it continuation prediction.



(a) Actor learning.



(b) Critic learning.

Figure 2.9: Actor-Critic Learning. We roll out imagined trajectories from posterior states and use the actor to pick actions. The value function is learned to predict the values during imagined rollouts as specified in the critic loss eq. (2.30). The actor is optimized to maximize the value targets, which is done by backpropagation through the dynamics model. The same imagined rollouts are used in actor learning and critic learning, but they are shown separately here to emphasize that the actor and critic train on their own parameters to maximize their own losses.

Chapter 3

Method

In this chapter, we describe the design of the Reinforcement Learning environment and the Path-Following and Collision Avoidance task to be solved. We introduce our observation function and reward function, how we chose the parameters of the reward function as well as some details about the experiment setup.

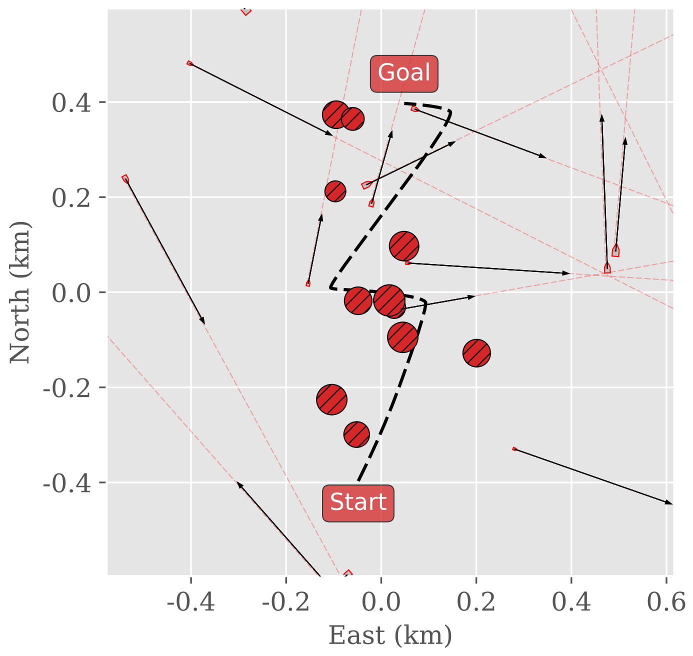


Figure 3.1: Illustration of the Reinforcement Learning environment. The environment consists of a randomly generated path, surrounded by randomly placed static circular obstacles, as well as dynamic obstacles moving in predetermined paths. A new, randomly generated scenario is trained on in each episode. As we see, static obstacles often lie in the path, while dynamic obstacles may cross the path, meaning a trivial solution such as a Line-of-sight heading autopilot alone is not sufficient to solve the environment reliably without collisions. Scenario and visualization code is courtesy of [26].

3.1 Observation function

Our observations o consist of two parts. The first is a low dimensional vector o^d consisting of vessel dynamics and path-following features from guidance, navigation and control theory. The second, o^r is a top-down image of the environment produced by a renderer of the simulation environment.[1]

$$o_t = [\underbrace{o_t^d}_{\text{Dense}}, \underbrace{o_t^r}_{\text{Image}}] \tag{3.1}$$

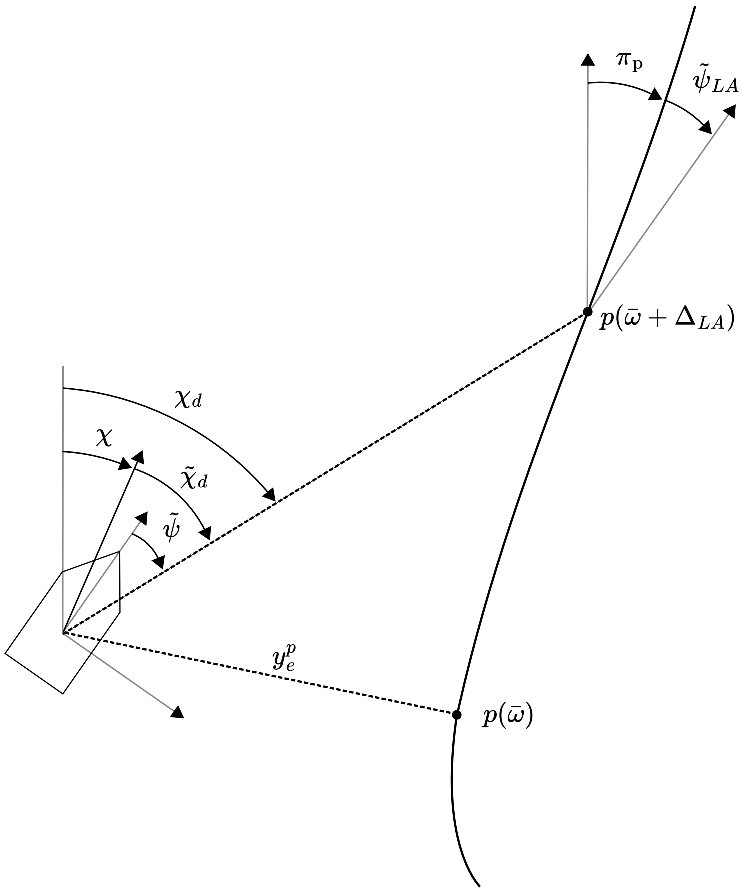


Figure 3.2: Path following states. In the figure, the course angle χ is not aligned with the heading angle ψ due to a non-zero sway velocity $v \neq 0$. The path angle π_p refers to the path angle at the look-ahead point $\mathbf{p}(\bar{\omega} + \Delta_{LA})$

[1]We use o^r for *renderer* rather than o^i for *image* to avoid confusion where i is seen as an index variable.

3.1.1 Dense navigation observations

Our observation function's dense component includes vessel dynamics and path-following features. The first six elements are similar to prior work by [26][21][22], while the addition of the path progression $\alpha_{\bar{\omega}}$ is novel. For completeness, we quickly recapitulate the navigation quantities from prior work here.

We let $\mathbf{p}(\omega) = [x_d^n, y_d^n]^T$ denote a path parameterized by ω. For a position in the NED frame, the closest point is then denoted by

$$\bar{\omega} = \arg\min_{\omega} \| [x^n, y^n]^T - p_d(\omega)^T \| \tag{3.2}$$

Further, we let the heading error $\tilde{\psi}$ be defined as the difference between the heading and the heading pointing towards the look-ahead point at $\mathbf{p}(\bar{\omega} + \Delta)$, as defined in eq. (3.3).

$$\tilde{\psi} = \text{atan2}\,(y_d(\bar{\omega} + \Delta_{\text{LA}}) - y^n,\, x_d(\bar{\omega} + \Delta_{\text{LA}}) - x^n) - \psi \tag{3.3}$$

Additionally, we also use the look-ahead heading error $\tilde{\psi}_{\text{LA}}$, which indicates the heading relative to the angle of the path at the look-ahead point eq. (3.4). The look-ahead heading error indicates the direction of the path further ahead, which may be useful when an obstacle needs to be circumvented, and the shortest detour is sought.

$$\tilde{\psi}_{\text{LA}} = \pi_{\text{p}}(\bar{\omega} + \Delta_{\text{LA}}) - \psi \tag{3.4}$$

$$\pi_{\text{p}}(\omega) = \text{atan2}\left(y_p'(\omega), x_p'(\omega)\right) \tag{3.5}$$

We also define the signed cross-track error, y_e^p as

$$y_e^p = -(x^n - x_p^n(\bar{\omega}))\sin(\pi_{\text{p}}(\bar{\omega})) + (y^n - y_p^n(\omega))\cos(\pi_{\text{p}}(\bar{\omega})) \tag{3.6}$$

As discussed in section 2.4.7, Dreamer predicts continuation flags $\hat{c}_{1:T}$ in imagined trajectories during actor-critic-learning. These are used to disregard rewards received after the predicted end of an episode when estimating values. The episode ends when $\alpha_{\bar{\omega}} > 0.99$ or the vessel is within a certain radius of the final waypoint. From the first six features in table 3.1 alone, predicting the end of the episode would be hard. Therefore, we include the progress $\alpha_{\bar{\omega}}$ to help the model's continuation predictions. We note that including this as a feature may be redundant when also using image observations as the agent may see that the path ends, but it should nevertheless increase the accuracy of these predictions.

We let $\bar{\varpi}$ denote the path length parameter of the closest point on the path, while ω_{end} denotes the path length at the goal at the end of the path. Then, $\alpha_{\bar{\varpi}} \in [0,1]$ denotes the path progress

$$\alpha_{\bar{\varpi}} = \frac{\bar{\varpi}}{\omega_{\text{end}}} \tag{3.7}$$

Symbol	**Name**
u	Surge velocity
v	Sway velocity
$\dot{\psi}$	Yaw rate
$\tilde{\psi}$	Heading error
$\tilde{\psi}_{LA}$	Heading error relative to look-ahead point
y_e^p	Cross-track error
$\alpha_{\bar{\varpi}}$	Path progress

Table 3.1: Navigation features used in o^d.

3.1.2 Renderer image observations

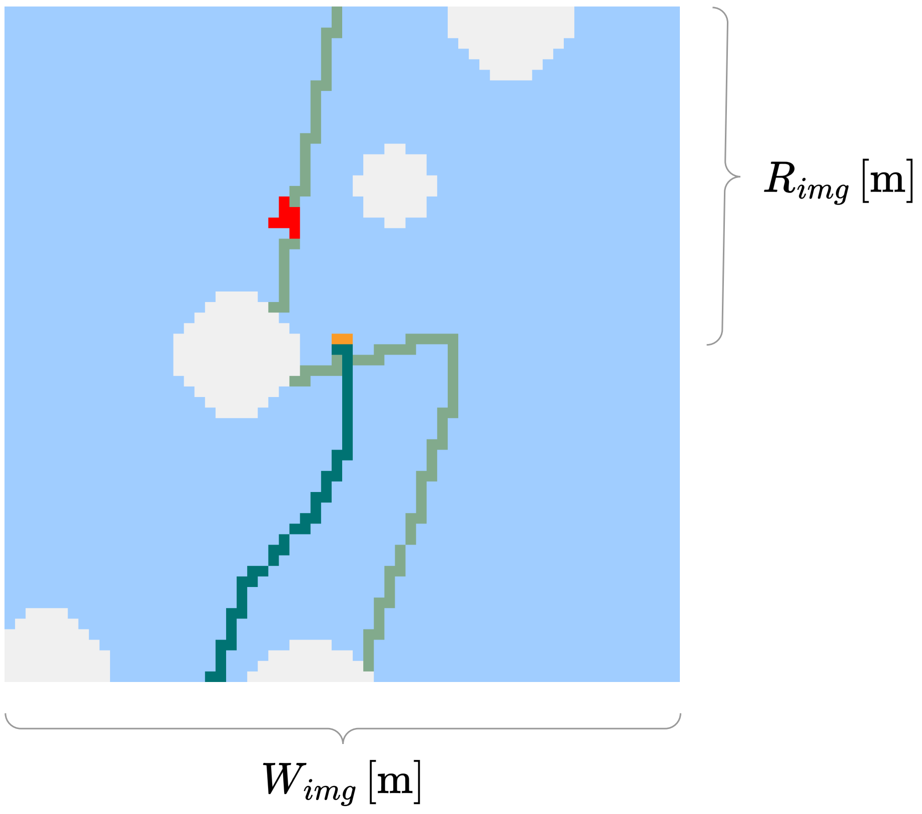


Figure 3.3: Renderer image observation

Color	Name
⬜	Static obstacle
🟥	Vessel obstacle
🟧	Own vessel
🟦	Own vessel trajectory so far
🟩	Path
🟦	Traversable water

Table 3.2: Color encoding of objects in image.

DreamerV1 [14] and DreamerV2 [16] were constructed for use with image observations. Therefore, we provide a renderer observation from the simulation framework to create a 64×64 RGB image as a part of the observation.

In fig. 3.3, we see the image observations used in this work. The image is centered w.r.t. the agent such that the bow of the vessel always points towards the top of the image, regardless of the heading angle ψ. We encode the obstacles, path, and water using colors according to table 3.2. In our experiments, we use the underlying state of the environment to provide these images.

We note that this approach is similar to a (non-probabilistic) occupancy grid [9] incorporating information about both obstacles and the task (i.e., the path to be followed) in the environment. We use $W_{img} \times W_{img} = 64 \times 64$ pixel RGB images, which shows an area of $2R_{img} \times 2R_{img} = 320m \times 320m$. This implies that each pixel spans $5m \times 5m$. If we want to increase either the range of the image or the accuracy, the number of pixels would need to be increased further.

3.2 Reward function

The reward function mathematically specifies the desired behavior of the agent. Its design is therefore critical for the agent to learn to balance the tasks of path-following and collision avoidance.

In our work, we use a reward function similar to the one proposed in [26] for the collision avoidance part but introduce a new path reward function.

3.2.1 Collision avoidance reward function

We use the same collision avoidance reward as in [26], but include a scaling coefficient $\beta_{\text{colav}} > 0$. The scaling coefficient is needed as we find that in addition to the relative trade off between r_t^{path} and r_t^{colav}, the absolute scale matters. We let $x_i \in (0,1)$ denote the closeness of an obstacle, while v_y is the component of the velocity towards the agent. The collision avoidance reward function component is then given by

$$r_t^{colav} = \beta_{\text{colav}} \frac{\sum_{i=1}^{N} \frac{\zeta(\theta_i)}{1+\gamma_\theta|\theta_i|} \exp(\gamma_v \max(0, v_y^i) - \gamma_g x_i))}{\sum_{i=1}^{N} \frac{\zeta(\theta_i)}{1+\gamma_\theta|\theta_i|}} \tag{3.8}$$

3.2.2 Path-following reward function

Prior work [26] [27] introduces the value function eq. (3.9), while [21] introduces eq. (3.10).

$$r_t^{\text{path,a}} = \left(\frac{u}{U_{max}} \cos\tilde{\psi} + \gamma_r \right) (\exp(-\gamma_\epsilon \mid \epsilon \mid) + \gamma_r) - \gamma_r^2 \tag{3.9}$$

$$r_t^{\text{path,b}} = \frac{u}{U_{\text{max}}} \frac{1+\cos\tilde{\psi}}{2} \frac{1}{|y_e^p|+1} \tag{3.10}$$

While both of these show good results, we argue that the path reward function can be simplified further. The path rewards are all based on line-of-sight guidance, which is meant to balance progression along the path while gradually decreasing the cross-track-error $|y_e^p|$. We define the course $\chi = \text{atan2}(\dot{y}^n, \dot{x}^n)$, and the desired course $\chi_d = \text{atan2}(y_d(\bar{\omega}+\Delta_{\text{LA}})-y^n, x_d(\bar{\omega}+\Delta_{\text{LA}})-x^n)$, which is directed towards the look-ahead point. We then propose a new, simpler path reward function based on minimizing the course error $\tilde{\chi} = \chi_d - \chi$.

$$r_t^{path} = \beta_{\text{path}} \frac{\sqrt{u^2+v^2}\cos\tilde{\chi}}{U_{max}} \tag{3.11}$$

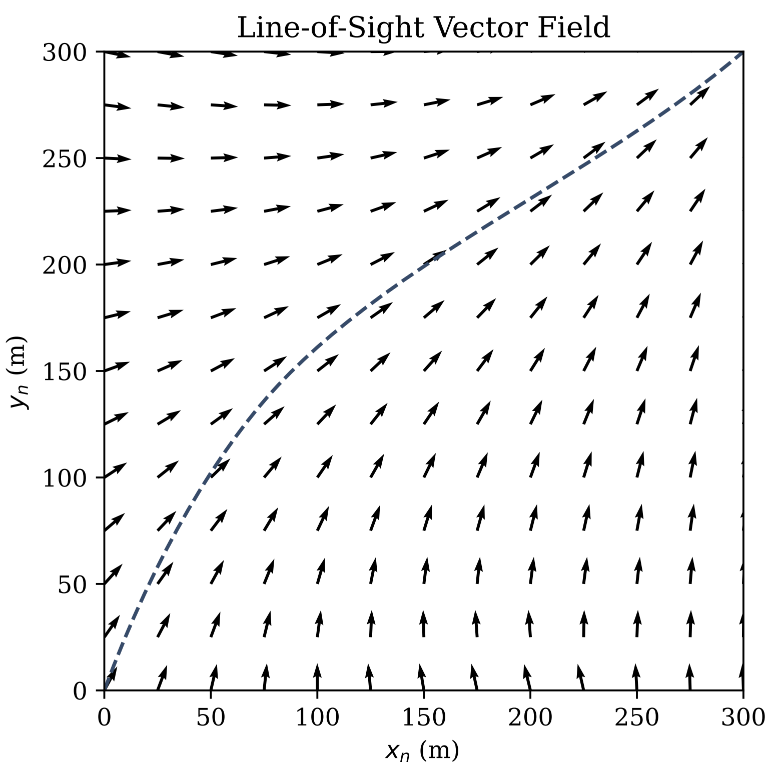


Figure 3.4: Plot showing the direction corresponding to $\tilde{\chi} = 0$ at different points around the path. A look-ahead distance of $\Delta = 150$ [m] is shown in the plot, and is also used in the experiments.

We include a tuning parameter $\beta_{\text{path}} > 0$, as well as a scaling parameter $\beta^{-}_{path} > 1$, which disincentives going backwards.[2] Then, we have that

$$\tilde{r}_t^{\text{path}} = \min\left(r_t^{\text{path}}, \beta^{-}_{\text{path}} r_t^{\text{path}})\right) \tag{3.12}$$

[2]Although not mentioned in [26], we find a similar scaling of negative rewards (also including closeness penalties) in the implementation.

3.2.3 Resulting reward function

Adding these together, this leads to the reward function

$$r_t^{\text{total}} = \begin{cases} r^{\text{collision}} & \text{if collision} \\ \lambda \tilde{r}_t^{\text{path}} + (1-\lambda) r_t^{\text{colav}} + r^{\text{exists}} & \text{otherwise} \end{cases} \tag{3.13}$$

However, we found that Dreamer often struggles with large magnitudes in the rewards. The original Dreamer agent operated on environments from the Deepmind Control Suite, where all rewards are in the range $r_t \in [0,1]$. DreamerV2 primarily used Atari environments, but limits the range of the rewards using a tanh-function. We found it necessary to also limit the rewards this way, leading to the rewards provided to the agent being given by

$$r_t = \tanh(r_t^{\text{total}}) \tag{3.14}$$

3.3 Reward function tuning using recorded trajectories

The reward function presented in section 3.2.2 includes many parameters that may be adjusted to encourage different behaviors. One approach that may come to mind is to train an agent, examine its behavior, and tune the reward function to see if the new agent exhibits the desired behavior. However, the feedback cycle in such an approach is slow. Depending on the problem, training an agent may typically take hours to days in wall-clock time[3] to perform. This makes tuning a reward function by training agents to maximize them an endeavor that may take days to weeks before being aligned with our interests.

Our environment is fairly easy for a human to solve. The action space is small, consisting of only two continuous inputs[4]. Further, our renderer provides a top-down view of the environment which is sufficient for a human player to navigate through the environment while avoiding obstacles. The reward function is also deterministic with respect to the state of the environment. Because of these factors, we may record the underlying state of our environment and tune the reward function afterwards without having to recreate the situations for every change.

In our work, we repeat these two steps until a reward function that suits our needs is acquired.

- **Collection of episodes through human play:** Using keyboard input and a rendering of the environment as output, we imitate different behaviors where we are interested in the corresponding reward.
- **Reward function tuning:** Using the recorded underlying state of our trajectories, we visualize the rewards of different classes to tune the reward function to be aligned with intended behavior.

DreamerV1 and DreamerV2 have numerical instability issues when dealing with large-magnitude rewards. In [26], a value of $r^{\text{collision}} = -10000$ was used. Although this clearly and simply specifies that collisions are undesirable, we find that it is not numerically stable to train Dreamer with large-magnitude rewards like this. Therefore, extra care should be taken to avoid misalignment between our intended goal and the one we specify in our reward function. As we showed in the previous section, we apply the $\tanh$-function per step to our rewards, such that the following equation holds

$$-\frac{\pi}{2} < r_t < \frac{\pi}{2} \tag{3.15}$$

[3]I.e. real-world run time of the training, not the amount of experience collected within the simulation environment.

[4]For comparison, the "Dog" tasks in Deepmind Control Suite [37] has a high-dimensional action space consisting of 38 continuous actions at every time step. Achieving close to optimal rewards for a human using keyboard inputs in these environments would be very challenging if not impossible.

3.3.1 Reward function tuning design choices

To tune our reward function while adhering to these limits, we need to design our reward function in a way that avoids exploits of the reward function. We propose tuning the reward function such that the returns $R(\cdot)$ satisfy

$$R(\tau_{\text{backwards}}) < R(\tau_{\text{circle}}) \approx R(\tau_{\text{still}}) < 0 < R(\tau_{\text{progress}}) < R(\tau_{\text{success}}) \tag{3.16}$$

Where we let the trajectories be classified according to table 3.3.

$\tau_{\text{backwards}}$	The agent moves in the opposite direction of the goal
τ_{circle}	The agent continuously moves in a circle
τ_{still}	The agent barely moves or stays completely still throughout the episode
τ_{progress}	The agent achieves progress towards the goal but isn't able to reach it due to collision or truncation
τ_{success}	The agent reaches the goal without colliding

Table 3.3: Trajectory classes of interest.

This classification is somewhat crude and doesn't consider the internal ranking of individual trajectories within the different classes. While also satisfying the inequalities above, the reward function should obviously also incentivize other desired behavior, such as staying close to the path and keeping a safe distance from obstacles while being efficient in reaching the goal.

To justify the inequalities above, we point out our hypotheses of some subtle exploits of what might happen during training if the inequalities above are *not* met in table 3.4.

We tune our reward function using these design choices and find the parameters shown in table 3.5.

Hypothetical scenario	**Possible exploit**
$R(\tau_{\text{progress}}) < 0$	If the return for progress along the path is negative, the agent may maximize the return by learning to end the episode by intentionally colliding, avoiding the additional negative rewards accumulated when trying to solve the task.
$R(\tau_{\text{progress}}) < R(\tau_{\text{circle}})$	The agent is unlikely to learn to reach the goal through chance alone. If continuously going in circles consistently gives a higher return than progressing toward the goal without reaching it, the agent is likely to try to exploit this behavior. This may then cause the agent to never learn to solve the task.

Table 3.4: Possible reward exploits if the inequalities in eq. (3.16) are not fulfilled.

Symbol	Value
γ_θ	10.0
γ_x	0.1
γ_{v_y}	1.0
$r_{\text{collision}}$	-500
λ	0.5
β_{path}	1.5
β^{-}_{path}	2.0
β_{colav}	0.2
r^{exists}	0.55

Table 3.5: Rewarder parameters used in experiments

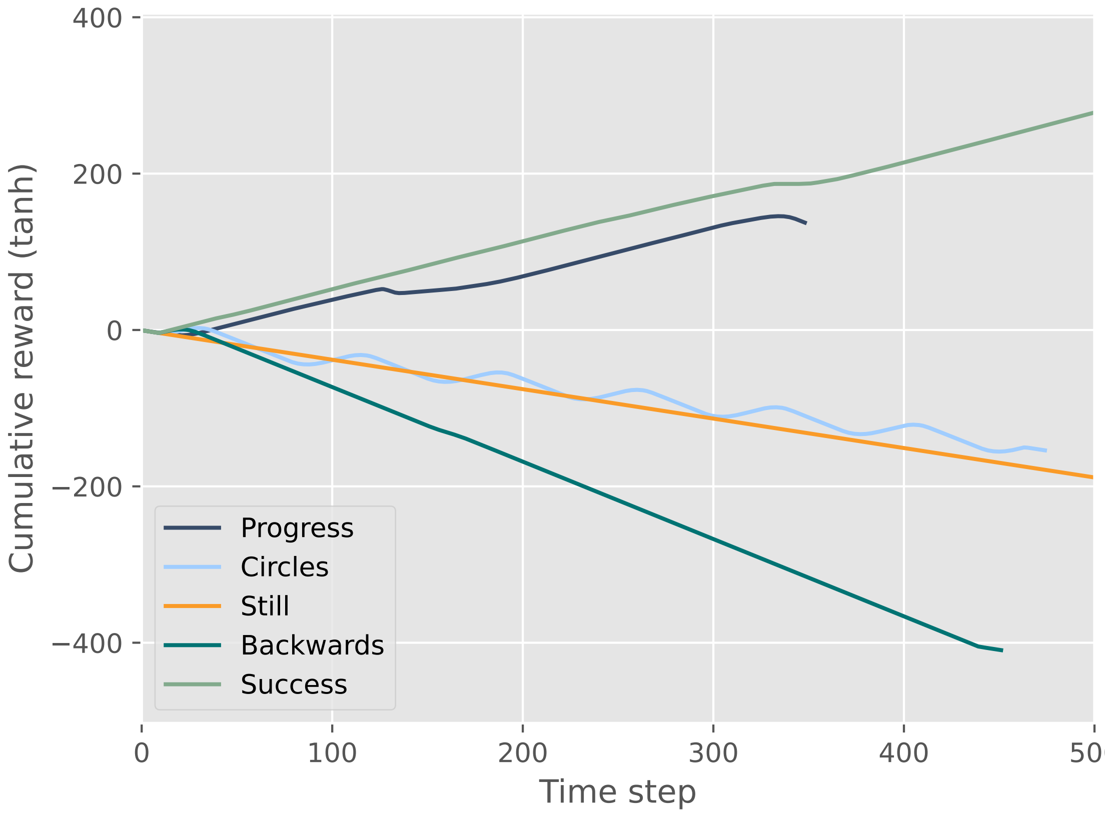


Figure 3.5: Cumulative rewards of some recorded trajectories after reward shaping. We use more trajectories than these to make sure we do not overfit our reward function to a few trajectories but only show a few here for clarity.

3.4 Termination and truncation

As the episodes may end in the real environment, we also need to be able to predict the continuation markers $c_{1:H}$ in imagined rollouts. In conventional Reinforcement Learning algorithms, the sampled data tells us when an episode terminates, i.e., there is no data from after it ends. In Dreamer, we imagine trajectories of the fixed length H during actor-critic learning. This has to be accounted for, which is done through the continuation model.

However, we wish for the imagined trajectories to only estimate the ends of episodes associated with a terminal event. This is also because we wish for the continue labels to be something that the agent can predict, which is hard to do when the conditions of truncation are not directly observable to the agent. There are two terminal events in our case, collisions and reaching the goal. Truncation is done if the episode is too long, or the cumulative reward is under a threshold. We point out that the environment rollout still ends, and we reset the environment if the episode is truncated. However, they are intentionally not marked as terminal events in the continuation markers c, s.t. the value function targets treat them like the episode goes on.

Situation	**Continue label c**
Collision	0
Reached goal	0
Episode return below threshold	1
Surpassed max time steps	1
All other time steps	1

Table 3.6: Labels used to train the continue predictor.

3.5 Encoder and Decoder Neural Networks

In our experiments, we use the standard architecture from DreamerV2 [16] to encoder our observations. The 2D encoder/decoder architecture is shown in fig. 3.6.

3.6 Experiment setup

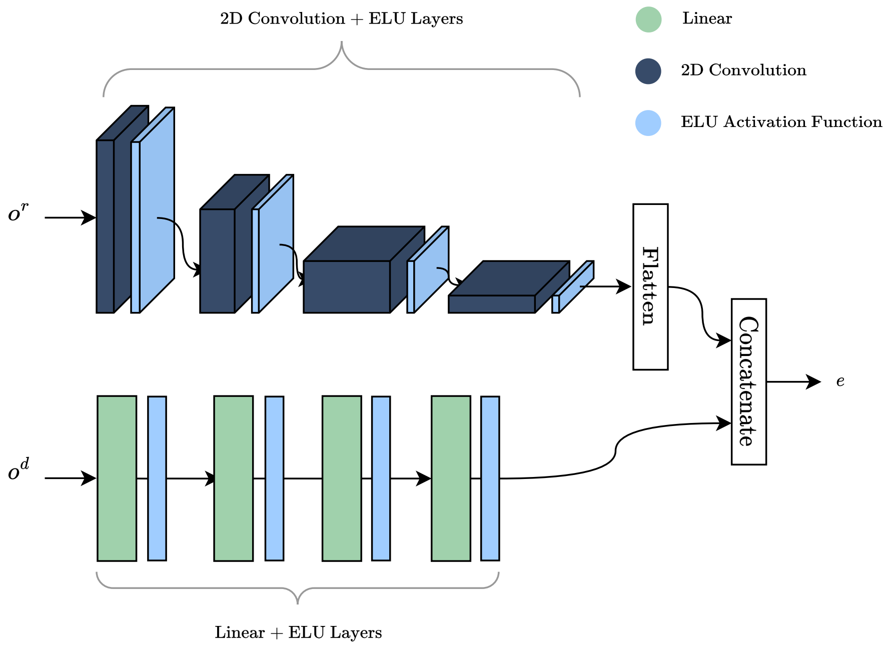
We summarize the configuration of our experiments in table 3.7.

Symbol/Name	**Value**	**Description**
α_ϕ	$1 \cdot 10^-5$	World model learning rate
α_ψ	$1 \cdot 10^-5$	Policy learning rate
α_ξ	$1 \cdot 10^-5$	Critic learning rate
B	50	Batch size
L	50	Batch length
H	15	Imagination horizon
P	50 000	Prefill timesteps
γ	0.995	Discount rate
λ	0.95	Value weighting parameter
n_{free}	1.0	Free nats
N_{deter}	200	Deterministic state size
N_{stoch}	200	Stochastic state size
T_{every}	10	Time steps between training iterations
β	1.0	KL-divergence loss scale

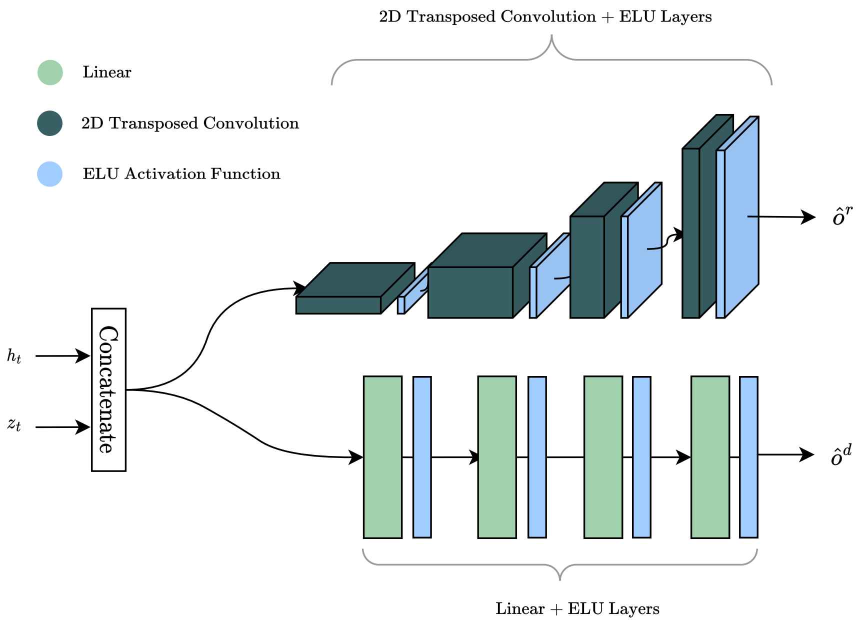
Table 3.7: DreamerV2 Hyperparameters used in experiments

Symbol	**Parameter**	**Value**	**Comment**
	Vessel model	Otter	Described in section 2.1
N_{rays}	Number of rangefinder rays	256	Used for reward calculation
Δt	Simulation time step	1 [s]	
$T_{\max}$	Maximum allowed time steps	2000 [s]	Episode is truncated if exceeded
$R_{\min}$	Minimum cumulative return	-2000	Episode is truncated if $R(\tau)$ falls below
Δ_{LA}	Look-ahead distance	150 [m]	
R_{img}	Image range	160 [m]	Allows seeing look-ahead point
W_{img}	Image height/width	64 [px]	

Table 3.8: Environment configuration used in experiments



(a) DreamerV2 2D encoder architecture.



(b) DreamerV2 2D decoder architecture.

Figure 3.6: Encoder/Decoder architectures. The convolutional network in the encoder decreases the size of the image while increasing the number of channels, which is concatenated with the features from the dense encoder to create the embedding e. The transposed convolutional network reconstructs the features from the latent state (h_t, z_t). The network uses the Exponential Linear Unit (ELU) activation function[8].

Chapter 4

Software Pipeline

In this chapter, we detail the software components used in our experiments. We also detail our efforts in extending the `gym-auv` simulation framework, which entailed over 5000 changed lines of source code. We also discuss our initial choice of using and extending an unofficial Dreamer implementation before we eventually decided to use the official implementation. The code used in this work is available on GitHub [6].

4.1 Simulation environment

The master thesis continues the work by Meyer [26] in `gym-auv`, and implements several improvements. The improvements include significant performance gains, as well as improvements in overall code quality. Notably, a new renderer has been written, which provides significantly cleaner and more extensible code. We also show our optimization procedure for the rangefinder sensors, which are not used as observations in our experiments but are still used to calculate the collision avoidance rewards.

4.1.1 Renderer

As our experiments depend on image observations, maintainable and extendable renderer code was a priority during our work. We found that the output of the existing renderer was mostly satisfactory, but the code was hard to read and modify. The previous renderer was based on the `pyglet` library, which we found hard to use as it interfaced directly into OpenGL for drawing simple geometric shapes. The new renderer is written using the `pygame` package. `pygame` is also used in the classical control environments in [4], and has a simpler interface for maintenance and expansion. This provides an interface that should feel familiar to developers in the Reinforcement Learning community.

The Renderer is based on 3 simple constructs. **Geometries** g represent the boundaries of obstacles, vessels or terrain. **Transformations** apply a combination of translations, rotations and scaling to a geometry and return a new geometry in another coordinate frame. **Factories** are pure functions that produce geometries from the environment state. A high-level overview of the rendering pipeline is shown in fig. 4.1

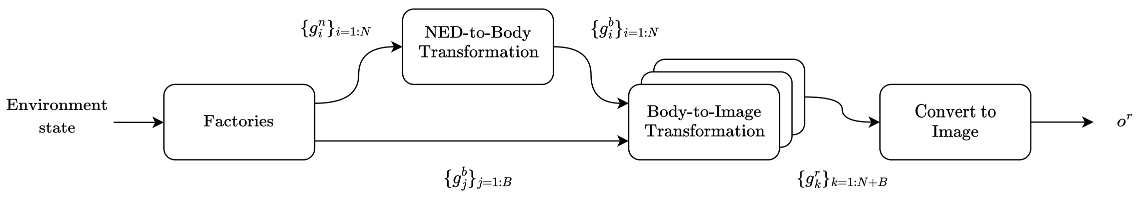


Figure 4.1: Summary of the structure of the renderer pipeline. The environment state is sent into factories, which make geometry objects for the agent vessel, obstacles, etc. Geometry objects in the NED-frame $\{g_i^n\}_{i=1:N}$ are converted into body-frame. Together with geometry objects in the body frame $\{g_i^b\}_{i=1:B}$, these are scaled, centered and rotated depending on the desired output into the rendering frame. These geometry objects are then converted into an RGB image o^r.

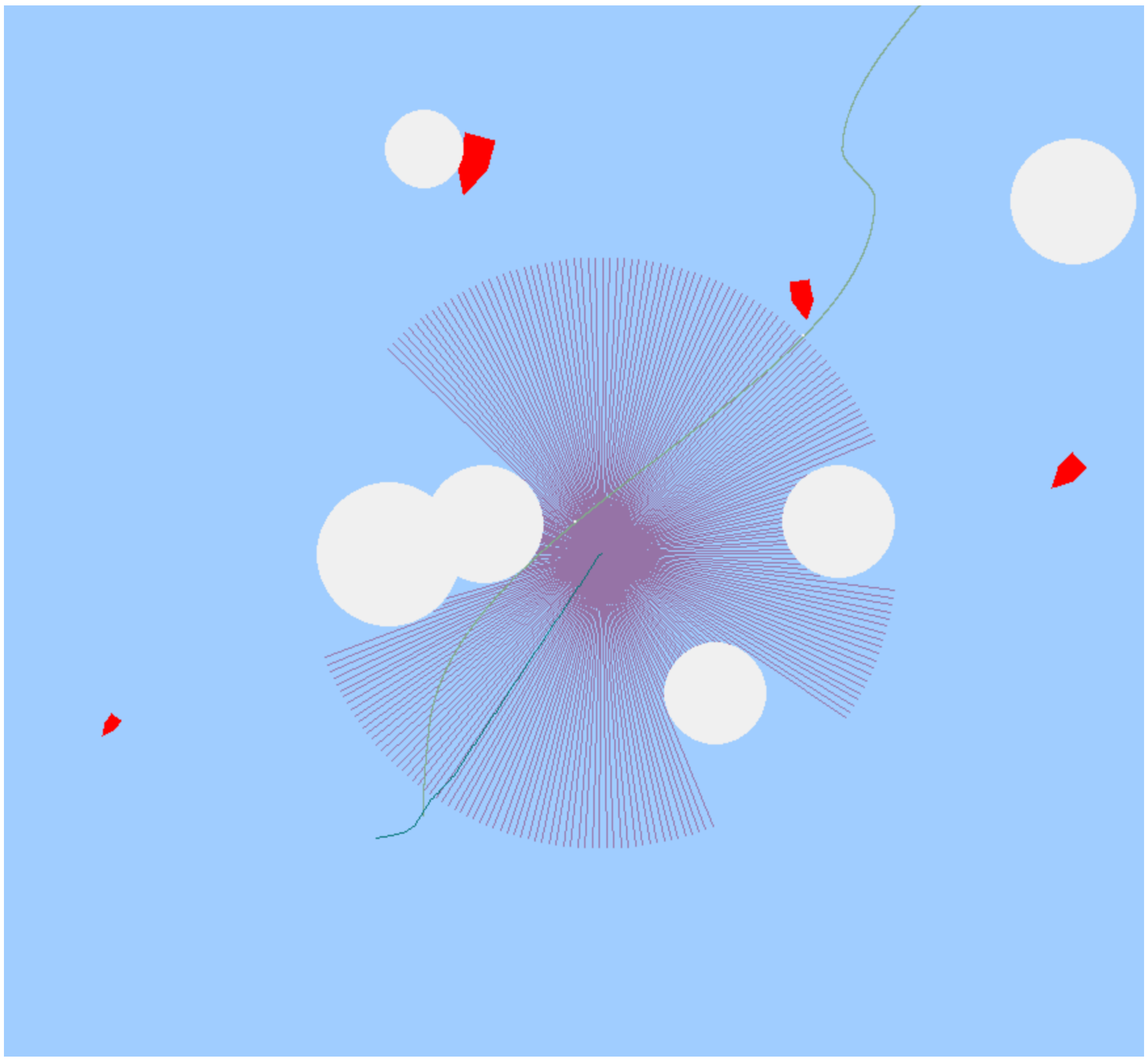


Figure 4.2: Our new renderer has a similar output to the original one in the environment but has a significantly cleaner code base. The renderer is implemented in a mostly stateless style, which we find easier to work with. This makes it simpler to reason about and avoids the complexities of circular dependencies with the environment, which the original suffered from. Simpler renderer code makes it easier to experiment with different image observations o^r.

4.1.2 Rangefinder simulator optimization using enclosing circles

Although we do not include rangefinder measurements in our observations, they are still used to calculate rewards in our experiments. Calculating the collisions between simulated sensor rays and obstacles is a major bottleneck of the `gym-auv` simulation framework. A run-time profiler found that almost all of the run-time was spent calculating the intersection between every ray and every obstacle. These expensive calculations were made regardless of whether it was geometrically possible that they may collide or not. This is a $\mathcal{O}(N_r N_o)$ operation, where N_r and N_o are the numbers of rays and obstacles respectively. To improve the run-time of the simulator, we optimize this bottleneck by pruning the search space.

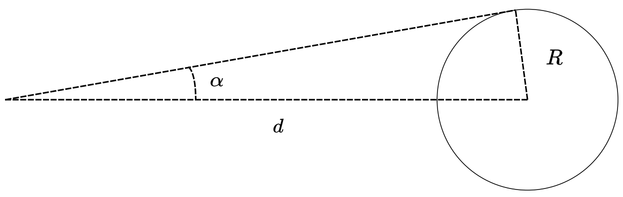
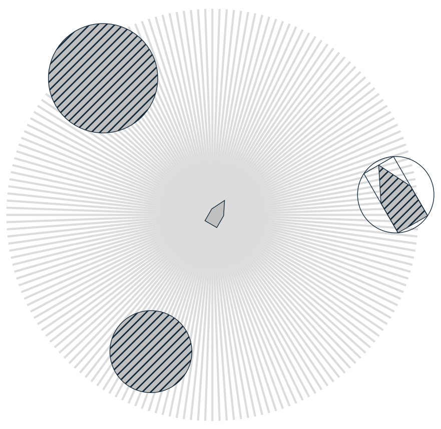
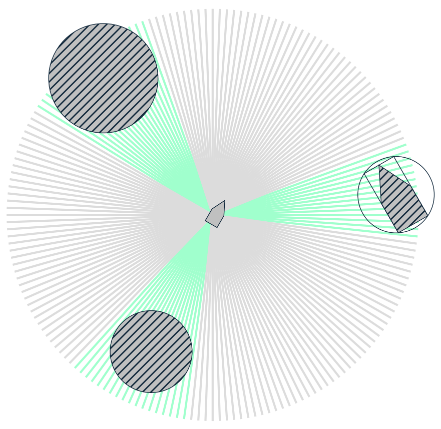
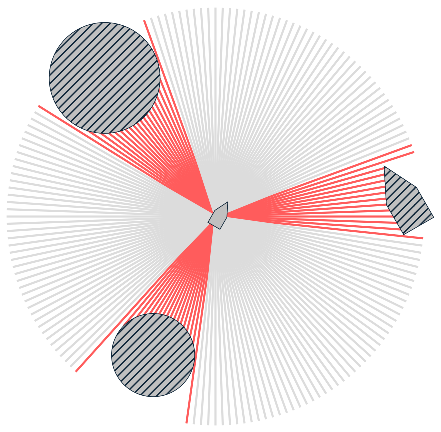


Figure 4.3: Figure showing the distance d, the radius R of an enclosing circle and the angle α. We can find the angle α using $\alpha = \sin^{-1} \frac{R}{d}$ and use it to find the rays that should be simulated for this obstacle.

We wish for our code to work for all shapes in the `shapely` package and use methods provided in the library. We decided to solve this in a 2-step process. We first find the minimum rotated rectangle, which is readily available for all shapes in the `shapely` library. Then, we find the enclosing circle as the one that touches all the corners of the minimum rotated rectangle. In the second stage, we find which rays should be simulated per obstacle. To accomplish this, we use some simple trigonometry, as shown in fig. 4.3. In the case where our rangefinder sensor is inside the enclosing circle (i.e., $|\frac{R}{d}| < 1$), we do not restrict the simulation of rays and simulate rangefinder ray collisions with all rays for the obstacle. Finally, the exact calculations for the ray intersections are made for the possible ray-obstacle pairs using the `shapely` library, as in the previous implementation. This means that while improving the run time of the rangefinder simulation, we have no loss of accuracy.


(a) Find enclosing circles for non-circular obstacles.


(b) Find rays that coincide with enclosing circles.


(c) Simulate only these rays.

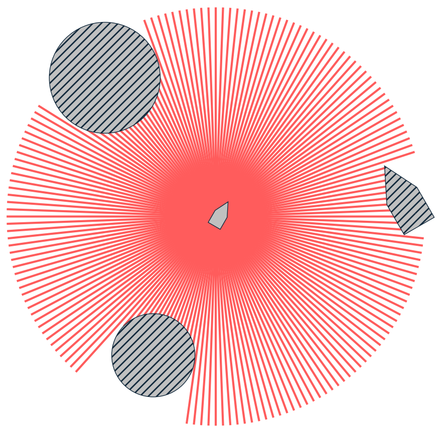

(d) Set other rays to max range.

Figure 4.4: Using a simple heuristic, we limit the search space of the rangefinder sensors, avoiding unnecessary computation.

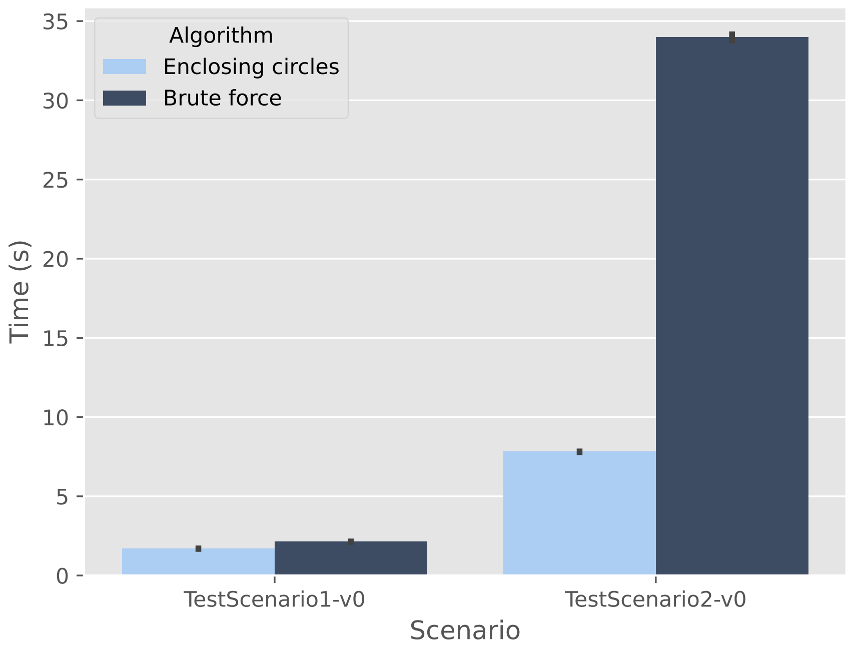


Figure 4.5: Our approach of pruning the search space with enclosing circles gives slight to significant speed-ups depending on the number of surrounding obstacles. Experiments have deterministic trajectories and are repeated 10 times. Testscenario1-v0 has on average 1 to 2 obstacles within the sensor range, while TestScenario2-v0 has 20 to 50. The "brute force" comparison does not make an approximation of the geometry as is done in [26], where it is referred to as a "virtual environment". The improved performance should allow the framework to scale to the use of more sensor rays and more complex terrain in the future when simulating rangefinders.

4.1.3 Other improvements to simulation framework

- **Increased type safety:** Our contributions significantly increase the type safety of the environment implementation. This makes further development of the code base significantly easier for several reasons. Firstly, type annotations make it possible for developer tooling (such as linters, code completion, etc) to easily understand the code. For instance, warnings are made about problems that may not be obvious to the developer, without the need of running the code. However, it is worth noting that Python does not check the type hints on its own as in strictly typed, compiled languages. The developer is responsible for using tooling that ensures type safety.

- **More robust numerical integration:** To solve the non-linear equations of motion, we use numerical integration methods from the `scipy` package. When changing our vessel model from the Cybership II to the Otter as described in section 2.1, we found the existing integration scheme to be numerically unstable with the new dynamics model. The prior approach included a custom Explicit Runge-Kutta scheme implementation to solve these equations. However, in our work, we use the `odeint` package in the `scipy` library to solve these equations. This approach has several advantages. Firstly, it simplifies the code by eliminating the need to implement our own integration method. Additionally, we improve the accuracy of the simulation by solving intermediate time steps.

4.2 Custom Dreamer algorithm implementation

Initially, we intended on using the rangefinder sensors as the measurements of our agent together with a 1D convolutional network. As the Dreamer implementations we found natively only supported dense or two-dimensional image observations, we wanted to find an implementation that was readable and easy to extend to use with 1D convolutions.

Multiple implementations of Dreamer exists, each with its own strengths and weaknesses. The most obvious one is the official implementation of DreamerV2 from the original author. We initially avoided this implementation, as it lacked some features and had a code base that we found hard to read. It often uses 1-character variable names, doesn't use type hints, and provides a very limited amount of comments.

At the beginning of our project, we spent much of our time customizing an implementation of Dreamer provided in Ray RLlib [23] to support the custom encoder and 1D convolutions. We found the code base in RLlib to be easier to read, and presumably easier to modify. It also had more mature utility functions, like experiment tracking using Weights & Biases [2], configurable checkpointing schemes, as well as hyperparameter optimization and experiment queueing and parallelization using Ray Tune [24].

During these experiments, we realized that although the Dreamer implementation in RLlib was an adaption of the original DreamerV1 agent, our finite-horizon episodic environment needed features from DreamerV2 to function properly. We decided to implement some of these features, such as continuation prediction, in our customized RLlib Dreamer implementation, which is summarized in table 4.1. Eventually, we eventually found a critical bug in the RSSM implementation provided in RLlib. The model didn't properly perform world-model learning, and considered its inputs as $B \cdot L$ one-step sequences rather than B sequences of length L. As we eventually questioned the overall correctness of the RLlib implementation and considered implementing the Dreamer algorithm out of the scope of this project, we stopped using it, instead focusing on the official implementation using image observations.

We have left out the experiments using 1D convolutions as we see them as inherently flawed due to the errors in the Dreamer algorithm implementation in RLlib we based these experiments on. They would not give a fair comparison to the experiments we present using image observations, as much of the differences would be caused by problems with the Dreamer algorithm implementation. However, we encourage further research to research using rangefinder sensors with 1D convolutional networks using the official implementation of DreamerV2 or DreamerV3.

Functionality	Original RLlib	Official DreamerV2	Custom (Ours)
Tensor library	PyTorch	TensorFlow	PyTorch
Image observations	**Supported**	**Supported**	**Supported**
Dense observations	Not supported	**Supported**	**Supported**
Custom models	Limited configurability	**Configurable**	**Configurable**
Utility functions	**Generalized**	Project-specific	**Generalized**
KL-balancing	Not available	**Available**	**Available**
Discount rate prediction	Not available	**Available**	**Available**

Table 4.1: Overview of features in our attempt to build upon the DreamerV1 implementation available in RLlib to build a version closer to DreamerV2.

Chapter 5

Experiments and results

The experiments were run on a 64-core AMD Ryzen Threadripper PRO 3995WX with 3 NVIDIA RTX A5000 GPU cards and 512 GB RAM. We use a single environment instance and one GPU per experiment. Although we experimented with a wide range of configurations during the project, the results in this section use the hyperparameters given in table 3.7. We also provide the training curves of the training run in our experiments in appendix B.

We focus our analysis on the following two research questions:

- **Task performance:** How well does Dreamer solve the task of path-following and collision avoidance?
- **World model prediction accuracy:** Is the model able to provide accurate, meaningful predictions about the future state of the environment?

5.1 Task performance

After 1.3 million time steps of training, we found the performance of the agent to have converged, and we measure the performance of the agent over 500 episodes. Our agent was successful in reaching the goal in 384 of the 500 episodes, giving a success rate of 76.8%. The average path progression of the episodes was 83.7%, giving the episodes that failed an average path progression of 29.7%[1]. In other words, they usually finished closer to the start of the path than the end, but make some progress along the path on average. The overall collision rate was 22.4%, meaning that almost all the failed episodes ended in a collision.

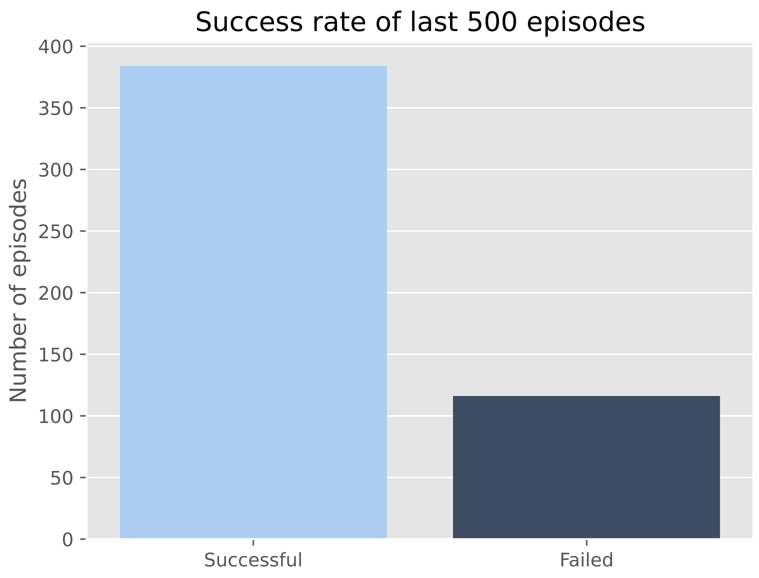


Figure 5.1: Success rate in reaching the goal without collision during 500 episodes.

[1]As the successful episodes have a path progression of 100%, the rest have a progression of $\frac{0.873-1\cdot 0.768}{1-0.768} = 0.297 = 29.7\%$

5.1.1 Agent trajectories

We present three trajectories performed by a trained agent.

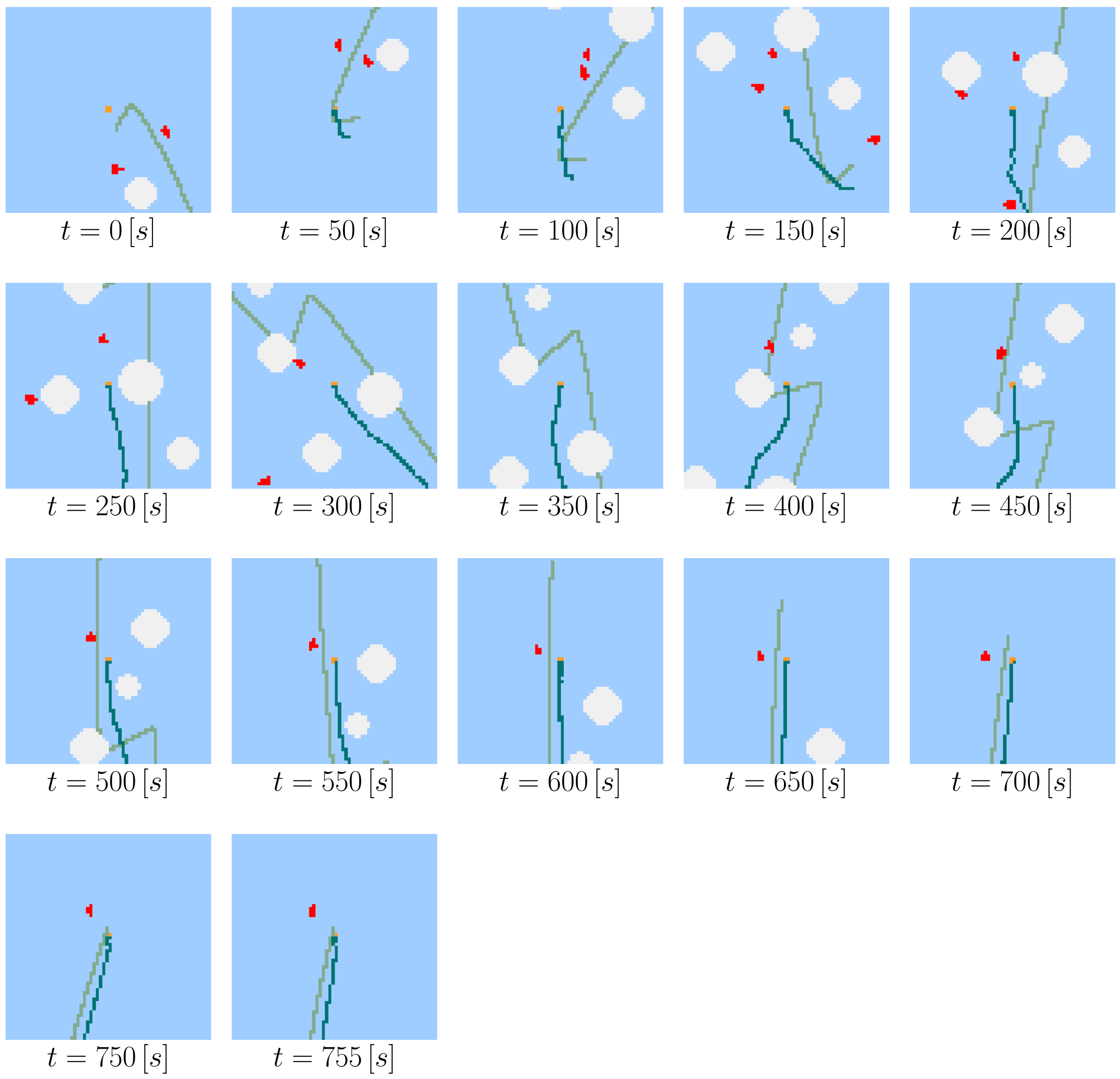


Figure 5.2: Example of a trajectory performed by a trained agent. The agent avoids the obstacles along the way and keeps a slight distance from another vessel near the end.

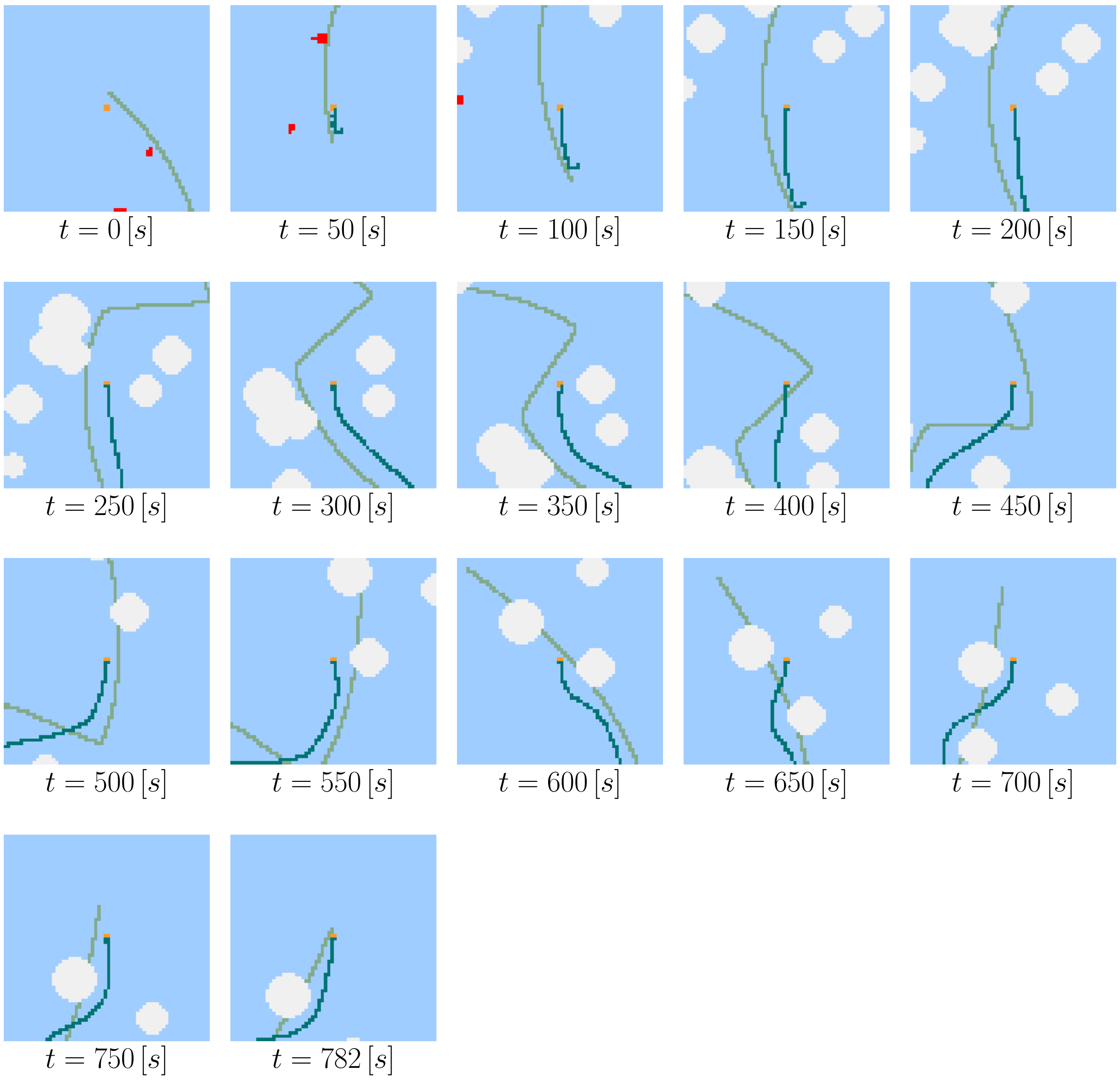


Figure 5.3: The trained agent successfully navigates through another environment.

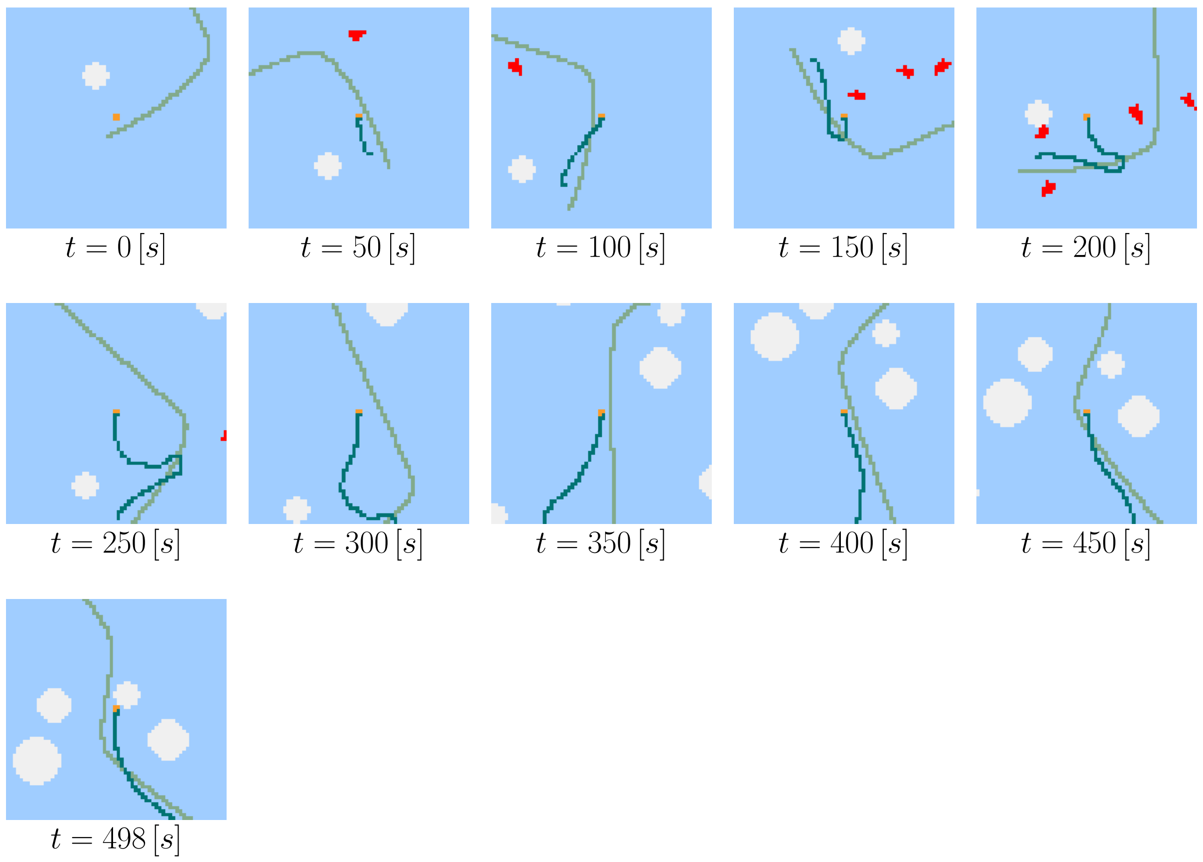


Figure 5.4: The agent manages to avoid multiple obstacles at the beginning, but eventually ends the episode by crashing into an obstacle near the path.

5.2 Reconstructions of imagined trajectories

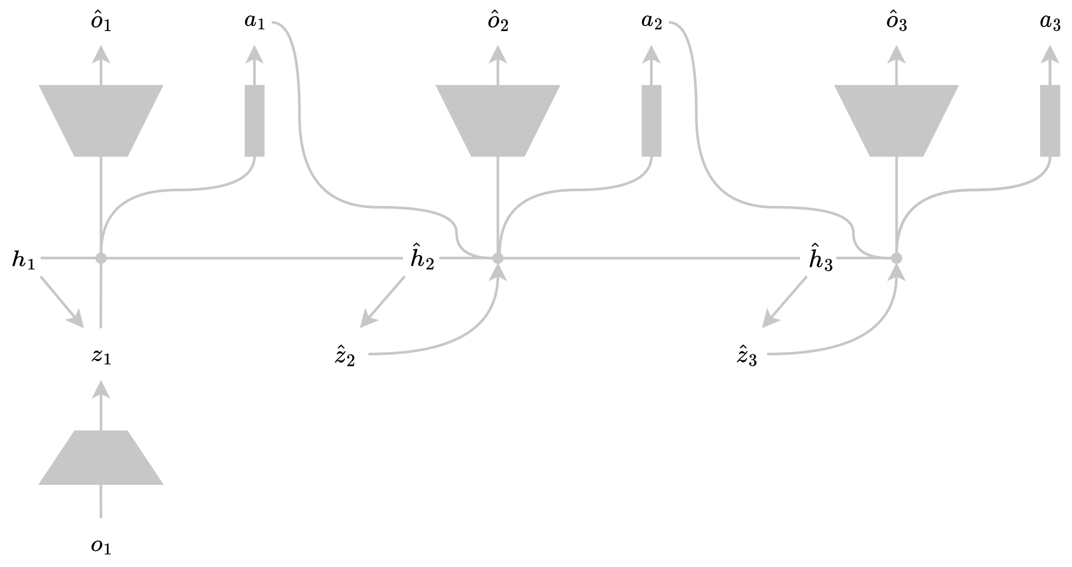


Figure 5.5: During the experiments done in section 5.2, we imagine trajectories from a short context of the first five observations and actions $(o_1, a_1), \ldots, (o_5, a_5)$, and use the decoder to predict the next 45 image observations $(\hat{o}_6, \ldots, \hat{o}_{50})$ given actions $(a_6, \cdots, a_{50})$. A context of one observation o_1 and prediction of three reconstructed imagined observations $(\hat{o}_1, \hat{o}_2, \hat{o}_3)$ given actions a_1, a_2, a_3 is shown in the figure.

5.2.1 Successful reconstruction

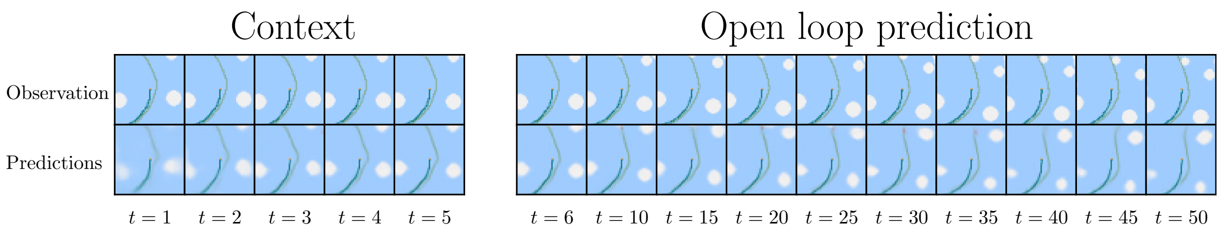


Figure 5.6: A mostly successful reconstruction.

First, we present an experiment where we find that reconstructing the observations works well. The agent is able to reconstruct the path, and is able to predict how its actions makes the agent move through the environment. Around $t = 15$, we see that a new obstacle appears in the actual observation. Since the existence of this obstacle was not known from the context $(o_1, \cdots, o_5)$, it is natural that the agent is not able to predict it.

5.2.2 Disappearing obstacles in reconstruction

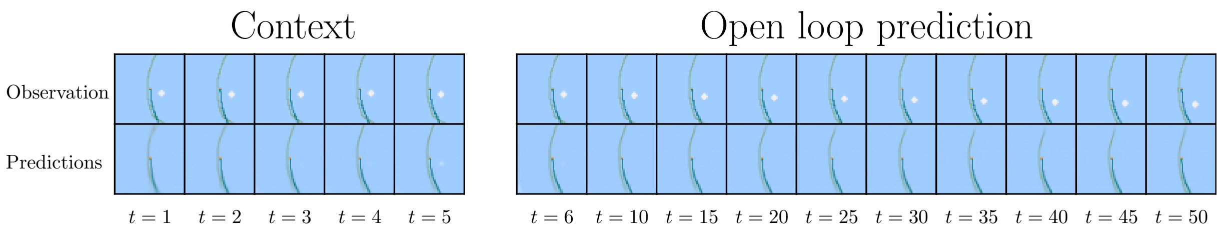


Figure 5.7: The white obstacle is absent from the reconstructions.

In another reconstruction using the same model, we find that the model does not consider the obstacle that is clearly present in the context $(o_1, \cdots, o_5)$. The reason for this is not completely clear, but the DreamerV3 paper [17], which was published shortly before the submission deadline of this work, provides a possible explanation. They mention that the visual complexity matters for how the hyperparameters should be tuned in DreamerV2. For instance, they claim that the KL-divergence coefficient β and the number of free nats should be tuned differently depending on whether the environment is more visually complex, such as in fig. 1.1, or if it depends more on individual pixels as in our experiments. DreamerV3 claims to have found a set of hyperparameters that work well across environments, but we leave experiments using this agent as further work.

5.2.3 Visual artifacts

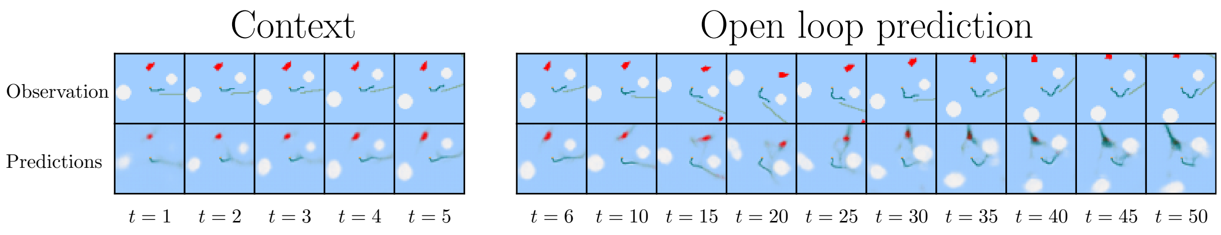


Figure 5.8: The reconstructions show visual artifacts around the red vessel.

In this reconstruction, we find that the model produces visual artifacts around the red vessel obstacle, which worsen towards the end of the predictions. The artifacts are most apparent for the vessel obstacle, which is understandable. The red vessel obstacle moves during the prediction horizon, while the white static obstacles stand still.

This problem is influenced by how these reconstruction experiments are set up. The stochastic nature of our experiment limits the predictive potential of our model. For instance, if we compare our experiments to the "Walker" or "Hopper" environments in fig. 1.1, there are many uncertainties in our environment. Our environment has randomly generated paths, as well as randomly placed static and moving obstacles. On the other hand, the "Hopper" and "Walker" environments do not have such uncertainties.

5.2.4 Consistency of results

One undesirable feature we found is that the performance of the agent is not always consistent. In another experiment with the same hyperparameters, reward function, observation function wildly different results were found.[2]The agent learned a policy similar to a "bang-bang" controller, providing max torque alternating between clockwise and anti-clockwise directions every step.

We hypothesize that this behavior filled the replay buffer (i.e., the training data set for world model learning) with trajectories that are harder to predict. No matter what is the cause of the poor reconstructions, we see an example of these reconstructions in fig. 5.9.

However, this other agent was often able to solve the task even though its reconstruction error was high. This other agent reached a success rate of 68.4% during the last 500 episodes of 1.85 million steps of training.

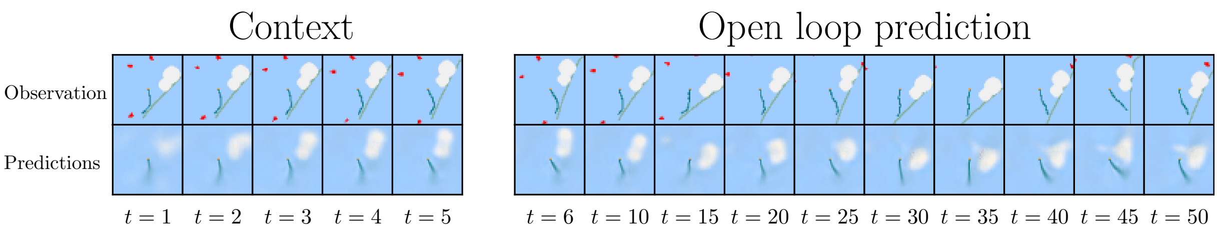


Figure 5.9: A typical reconstruction from a less performing training with the same hyperparameters and a similar amount of training. The white obstacles are generally blurry, and none of the vessel obstacles are reconstructed.

[2]There was one very slight change in the renderer, where we limit the dark blue trail of the agent to only show the last $100[s]$. We do not believe this is the sole cause of the differences, but rather hypothesize that differences between training runs (which we also saw during other experiments) caused the problems.

Chapter 6

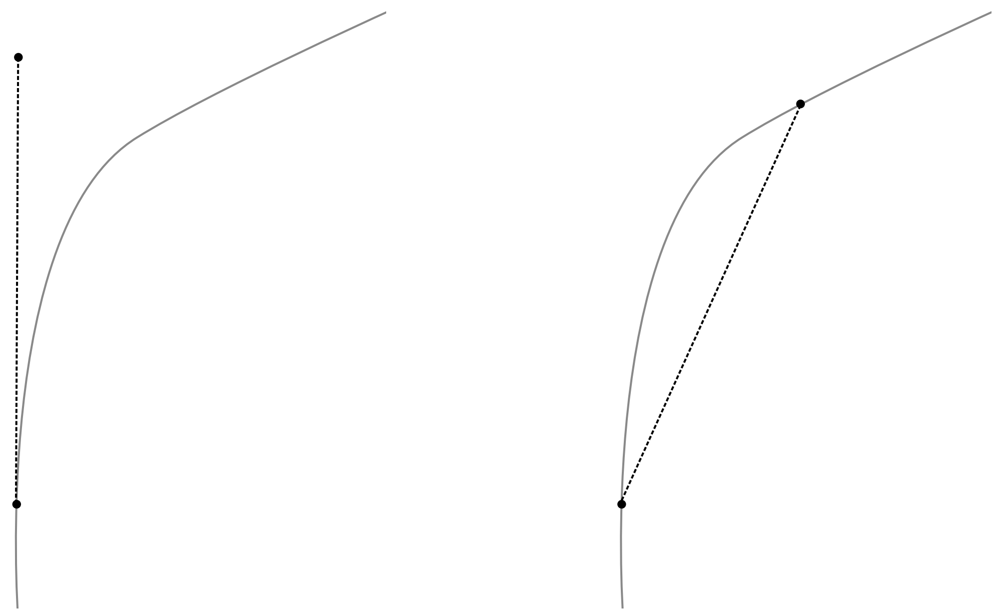
Discussion

6.1 Realism of access to renderer image observations

In our experiments, we create image observations using the underlying state of the environment. In a real system, the underlying state is not directly available to the agent. However, we argue that estimating these states in a real system is not entirely unreasonable. Autonomous vessels are often equipped with sensor fusion systems for target tracking and situational awareness, which may provide similar information. Additionally, static obstacles can often be known ahead of time through access to a map of the areas it operates in. Something similar to the renderer image may be constructed if information about obstacles' shapes, positions, and velocities as well as a map is available. However, our renderer image shows all information in the image and does not consider the occlusion of objects located behind other objects from the perspective of the agent.

6.2 Cross-track error and choice of look-ahead point

In section 3.2, we introduced a new and simpler reward function based on the product of speed and course error. Although we have confidence in that this formulation of the reward function has some merit, we believe that our choice of look-ahead point is not optimal. For instance, we see in fig. 5.3 that the agent consistently cuts the corners of the path. This behavior is incentivized in corners as moving straight towards the look-ahead point, which is on the other side of the corner, is optimal w.r.t. the path reward. When using Line-of-sight guidance for curved paths, [10] proposes the use of a look-ahead point along the path tangent rather than one along the path. Although our agent still mainly moves along the path, we recommend further work to experiment with using the path tangent as the look-ahead point for better cross-track performance.



(a) Path tangent look-ahead point.

(b) Along-path look-ahead point (ours).

Figure 6.1: Using the path-tangent to to provide the look-ahead point point may prevent the agent from cutting corners, thus reducing the cross-track error.

6.3 Model interpretability

Algorithm	Parameter Count
PPO (Meyer et. al. [26])	12.8 k
DreamerV2 (Ours)	18 M

Table 6.1: Approximate parameter counts for different reinforcement learning agents. The Dreamer agent has several orders of magnitude more parameters.

As we showed in section 5.2, our model-based reinforcement learning agent may predict future observations of its environment. While we believe that even more accurate predictions of the future may be possible and offer interpretability, these models still come with another trade-off. In table 6.1, we see the number of trainable parameters in our world model compared to the policy and value networks in prior work by [26]. Our model has more than 10000× the parameters of prior work, and it is likely infeasible to prove how such a model would behave well in all situations.

6.4 Misalignment of world-model loss function

We hypothesize that the world model loss may not be completely aligned with our objective of solving the task. As we see in eq. (2.28), the reconstruction losses for image-observations and reward functions are mean-squared errors. At every time step, there are many more pixels in the image than the single scalar reward, and they have similar orders of magnitude due to limiting the range of the reward using the $\tanh$-function. This may cause the world-model to implicitly prioritize predicting the pixels of the image well on average rather than learning features that are important to the task. A similar problem is mentioned in the DreamerV2 paper [16]. They point out that the algorithm performs poorly in the "Video Pinball" game, where the most important object (the ball) occupies only a single pixel in the image.

Chapter 7

Conclusions and Recommendations for Further Work

7.1 Summary and Conclusions

In our work, we show how a path-following and collision avoidance task may be framed for use with the model-based DreamerV2 Reinforcement Learning agent. We show how one might incorporate both path-following features as a vector and an overview of the situation as a low-resolution image and use these to navigate through a challenging environment with a success rate of 76.8%. We have also shown a method for tuning our reward function, and made significant improvements to the `gym-auv` simulation environment.

From our experiments, we find that the agent is often able to choose intelligent behavior in the environment, but is seemingly also limited by the world model's failure rate of correctly perceiving obstacles. As DreamerV3[17], the newest version of the Dreamer agent claims to provide hyperparameters that work well across a wide range of environments, we believe that our setup has the potential for higher performance in further work.

7.2 Recommendations for Further Work

7.2.1 Model-based Reinforcement Learning using Rangefinder sensors

In this work, we showed how we might adapt the simulation environment for use with the standard DreamerV2 [16] agent. However, one could instead use the rangefinder measurements with a 1D Convolutional encoder and decoder. Such a sensor setup makes fewer assumptions about the access to the environment state than our image observations and may be more applicable in practice. We point out that using rangefinder or image observations is not mutually exclusive. Multiple observation modalities may be encoded and concatenated in parallel, as we do with dense and image observations in fig. 3.6.

7.2.2 New tasks using DreamerV3

DreamerV3 [17] is the follow-up paper of DreamerV2. It focuses on improving the robustness and scalability of the model. DreamerV3 robustifies the agent to work with diverse environments w.r.t. reward scaling and observation fidelity, solving a wide range of tasks with the same hyperparameters. They also show favorable scaling properties of the model, with larger models showing higher data efficiency and capability. The DreamerV3 paper was released close to the deadline of this work, but the implementation is currently not available. We leave applying DreamerV3 to the `gym-auv` environment as well as other environments UAVs, quadcopter drones or quadrupedal robots as further work.

7.2.3 Kinematics-informed World Models

Although randomly initialized neural networks may learn complex non-linear functions, they do not possess our prior knowledge of physics. While all the papers written about Dreamer and its predecessor PlaNet use the RSSM model, we might also use these algorithms with other models.

Thor I. Fossen models Rigid-body systems such as marine vessels in their most general form using the following equations [10]

$$\text{Kinematics:} \quad \dot{\eta} = \mathbf{J}_{\boldsymbol{\Theta}}(\eta)\nu \tag{7.1}$$

$$\text{Kinetics:} \quad \mathbf{M}\dot{\nu} + \mathbf{C}(\nu)\nu + \mathbf{D}(\nu)\nu + \mathbf{g}(\nu) + \mathbf{g_0} = \tau \tag{7.2}$$

These equations differ as the kinetics is an approximated model, while the kinematics are exact. The first equation describes the kinematics of the system, i.e. how the velocities and angular velocities ν relate to the derivative $\dot{\eta}$ through the matrix $\mathbf{J}_{\boldsymbol{\Theta}}$. However, the second equation is slightly more complex, describing the kinetics of the system. Kinetics relate the forces and moments of the environment to $\dot{\nu}$, the acceleration and angular acceleration vector.

The main takeaway from this is that one might use the exact kinematics relation in a model while approximating the kinetics using a neural network. Rearranging the kinetics equation to isolate $\dot{\nu}$ on the left side and model of accelerations and angular accelerations from the physics of on the right-hand side, we get:

$$\dot{\nu} = \mathbf{M}^{-1}(-\mathbf{C}(\nu)\nu - \mathbf{D}(\nu)\nu - \mathbf{g}(\nu) - \mathbf{g_0} + \tau) \tag{7.3}$$

We hypothesize that $\dot{\nu}$ may be learned jointly by a neural network f_θ, such that the equations reduce to:

$$\dot{\eta} = \mathbf{J}_\Theta(\eta)\nu \tag{7.4}$$

$$\dot{\nu} = f_\theta(\eta, \nu, h, a) \tag{7.5}$$

To implement these equations in practice, they must be discretized. One might do this by explicit Runge-Kutta methods, or through simpler approaches such as Euler integration. Then, the predictions $\hat{\eta}$ and $\hat{\eta}$ according to the model are given by:

$$\hat{\eta}[k+1] \approx (\mathbf{I} + \mathbf{J}_\Theta(\hat{\eta}[k])) \cdot \Delta T)\,\hat{\nu}[k] \tag{7.6}$$

$$\hat{\nu}[k+1] \approx f_\theta(\hat{\eta}[k], \hat{\nu}[k], \hat{h}[k], a[k]) \cdot \Delta T \tag{7.7}$$

Importantly, if implemented in an auto-differentiation framework such as TensorFlow or PyTorch, this model is still differentiable. Assuming we have access to good ground-truth labels η_{true} and ν_{true} as well as the actions a performed, we might train the network by minimizing the distance between the predictions and ground truth values by optimizing the parameters of the kinetics prediction network, which are denoted by θ, that is:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} ||\hat{\eta}[k+1] - \eta_{true}[k+1]||^2 + ||\hat{\nu}[k+1] - \nu_{true}[k+1]||^2 \tag{7.8}$$

Doing this, we get a model closely resembling the rigid-body system that might be used to optimize over reward functions using these states. This model architecture might be used to solve tasks by training an actor-critic-style agent differentiating through the dynamics as in Dreamer, MPC style planning with the Cross-Entropy Method as in PlaNet [15], or a hybrid as is given in TD-MPC [18].

Appendix A

Acronyms

ASV Autonomous Surface Vehicle

COLREG Convention on the International Regulations for Preventing Collisions at Sea

CPU Central Processing Unit

GAN Generative Adversarial Network

GAE Generalized Advantage Estimation

GNC Guidance, Navigation, and Control

GPU Graphics Processor Unit

KL Kullback-Leibler (Divergence)

ML Machine Learning

MPC Model Predictive Control

PlaNet Planning Network

PPO Proximal Policy Optimization

RL Reinforcement Learning

RL Reinforcement Learning

RSSM Recurrent State Space Model

TD Temporal Difference

TRPO Trust Region Policy Optimization

TPU Tensor Processing Unit

VAE Variational AutoEncoder

Appendix B

DreamerV2 Training Curves

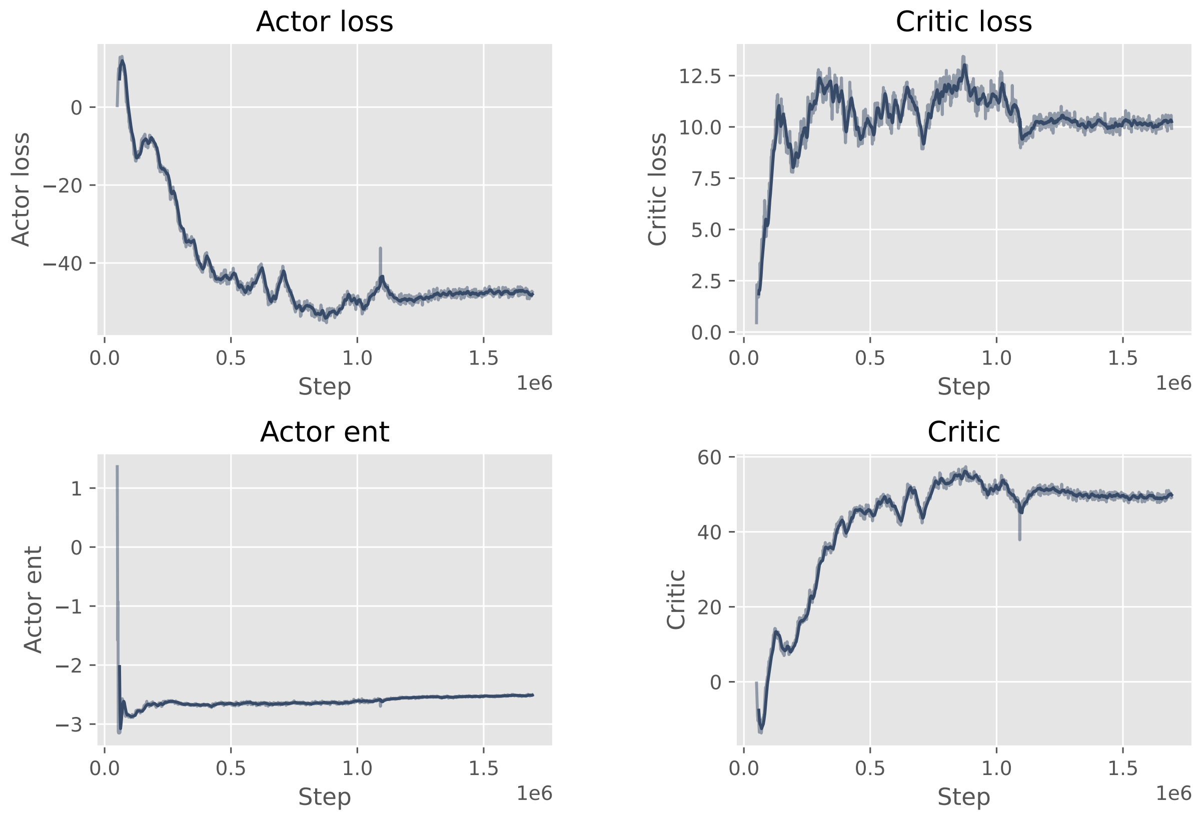


Figure B.1: Dreamer V2 training curves (1/2).

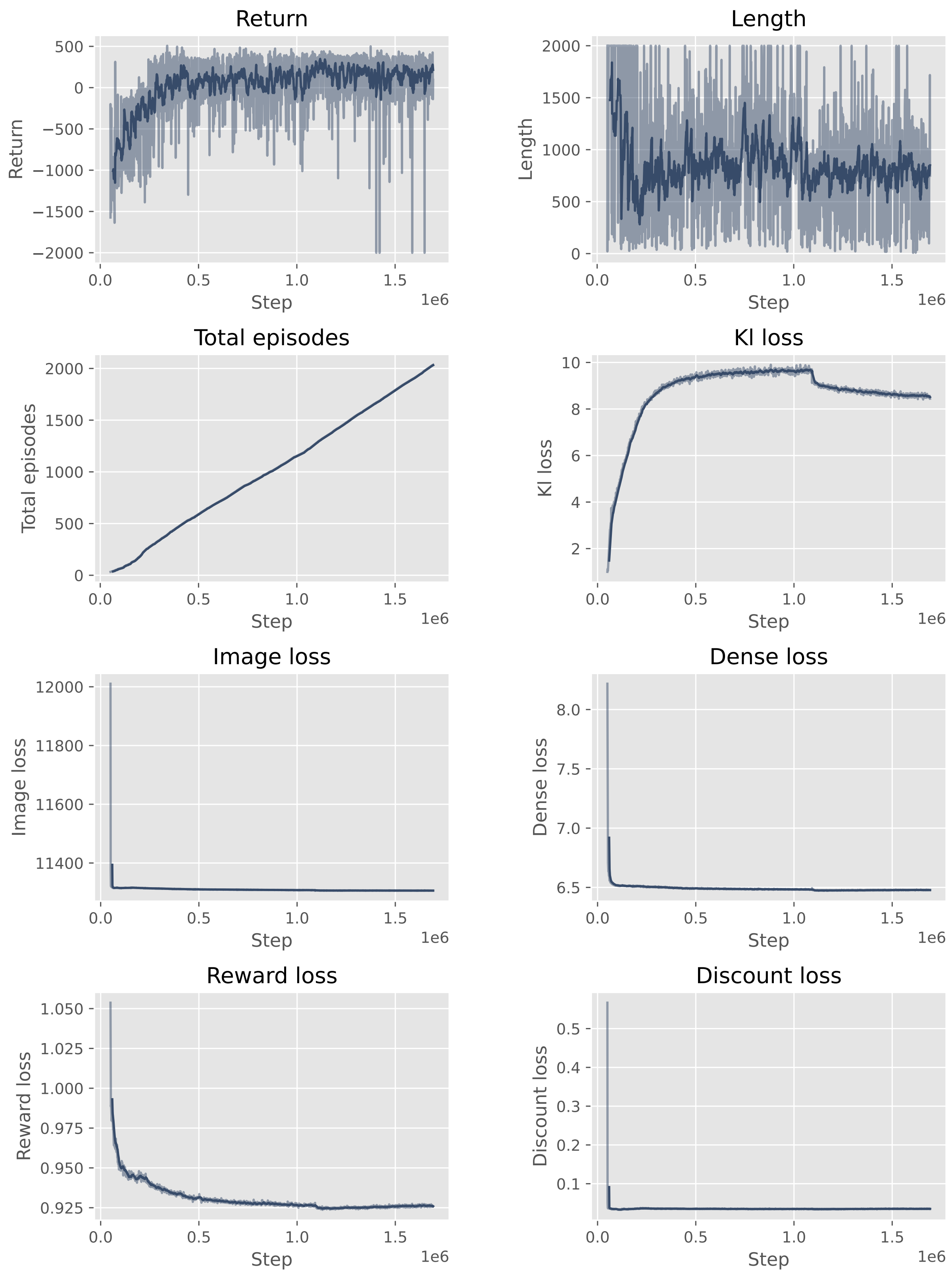


Figure B.2: Dreamer V2 training curves (2/2).

Bibliography

[1] Joshua Achiam. “Spinning Up in Deep Reinforcement Learning”. In: (2018).

[2] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: `https://www.wandb.com/`.

[3] Edmund F Brekke et al. “milliAmpere: An Autonomous Ferry Prototype”. In: *Journal of Physics: Conference Series* 2311.1 (July 2022), p. 012029. DOI: `10.1088/1742-6596/2311/1/012029`. URL: `https://doi.org/10.1088/1742-6596/2311/1/012029`.

[4] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[5] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165 (2020). arXiv: `2005.14165`. URL: `https://arxiv.org/abs/2005.14165`.

[6] Kristian Brudeli. *Master Thesis Github repository*. 2023. URL: `https://github.com/krisbrud/master-thesis`.

[7] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078 (2014). arXiv: `1406.1078`. URL: `http://arxiv.org/abs/1406.1078`.

[8] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: `http://arxiv.org/abs/1511.07289`.

[9] Thomas Collins, J.J. Collins, and Donor Ryan. “Occupancy grid mapping: An empirical evaluation”. In: *2007 Mediterranean Conference on Control Automation*. 2007, pp. 1–6. DOI: `10.1109/MED.2007.4433772`.

[10] Thor I Fossen. *Handbook of marine craft hydrodynamics and motion control*. Wiley, June 2021.

[11] Thor I Fossen. *Python Vehicle Simulator*. 2020. URL: `https://github.com/`.

[12] Florian Fuchs et al. “Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning”. In: *CoRR* abs/2008.07971 (2020). arXiv: `2008.07971`. URL: `https://arxiv.org/abs/2008.07971`.

[13] Danijar Hafner. *Github Issue: Lambda Target Equation*. 2022. URL: https://github.com/danijar/dreamerv2/issues/25#issuecomment-949729470.

[14] Danijar Hafner et al. "Dream to Control: Learning Behaviors by Latent Imagination". In: *arXiv preprint arXiv:1912.01603* (2019).

[15] Danijar Hafner et al. "Learning Latent Dynamics for Planning from Pixels". In: *CoRR* abs/1811.04551 (2018). arXiv: 1811.04551. URL: http://arxiv.org/abs/1811.04551.

[16] Danijar Hafner et al. "Mastering Atari with Discrete World Models". In: *CoRR* abs/2010.02193 (2020). arXiv: 2010.02193. URL: https://arxiv.org/abs/2010.02193.

[17] Danijar Hafner et al. *Mastering Diverse Domains through World Models*. 2023. DOI: 10.48550/ARXIV.2301.04104. URL: https://arxiv.org/abs/2301.04104.

[18] Nicklas Hansen, Xiaolong Wang, and Hao Su. *Temporal Difference Learning for Model Predictive Control*. 2022. DOI: 10.48550/ARXIV.2203.04955. URL: https://arxiv.org/abs/2203.04955.

[19] Simen Theie Havenstrøm, Adil Rasheed, and Omer San. "Deep Reinforcement Learning Controller for 3D Path-following and Collision Avoidance by Autonomous Underwater Vehicles". In: *CoRR* abs/2006.09792 (2020). arXiv: 2006.09792. URL: https://arxiv.org/abs/2006.09792.

[20] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (2021), pp. 583–589. DOI: 10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[21] Thomas Larsen et al. "Comparing Deep Reinforcement Learning Algorithms' Ability to Safely Navigate Challenging Waters". In: *Frontiers in Robotics and AI* 8 (Sept. 2021). DOI: 10.3389/frobt.2021.738113.

[22] Thomas Nakken Larsen et al. "Risk-based implementation of COLREGs for autonomous surface vehicles using deep reinforcement learning". In: *CoRR* abs/2112.00115 (2021). arXiv: 2112.00115. URL: https://arxiv.org/abs/2112.00115.

[23] Eric Liang et al. "Ray RLLib: A Composable and Scalable Reinforcement Learning Library". In: *CoRR* abs/1712.09381 (2017). arXiv: 1712.09381. URL: http://arxiv.org/abs/1712.09381.

[24] Richard Liaw et al. "Tune: A Research Platform for Distributed Model Selection and Training". In: *CoRR* abs/1807.05118 (2018). arXiv: 1807.05118. URL: http://arxiv.org/abs/1807.05118.

[25] Ludvig Løken Sundøen. *Path following and collision avoidance for quadcopters using deep Reinforcement Learning*. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3022408.

[26] Eivind Meyer. *On Course Towards Model-Free Guidance: A Self-Learning Approach To Dynamic Collision Avoidance for Autonomous Surface Vehicles*. 2020. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2780874.

[27] Eivind Meyer et al. "COLREG-Compliant Collision Avoidance for Unmanned Surface Vehicle Using Deep Reinforcement Learning". In: *IEEE Access* 8 (2020), pp. 165344–165364. DOI: 10.1109/ACCESS.2020.3022600.

[28] Eivind Meyer et al. *Taming an autonomous surface vehicle for path following and collision avoidance using deep reinforcement learning*. 2019. DOI: 10.48550/ARXIV.1912.08578. URL: https://arxiv.org/abs/1912.08578.

[29] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[30] OpenAI. *OpenAI Five*. https://blog.openai.com/openai-five/. 2018.

[31] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. DOI: 10.48550/ARXIV.2204.06125. URL: https://arxiv.org/abs/2204.06125.

[32] Robin Rombach et al. *High-Resolution Image Synthesis with Latent Diffusion Models*. 2021. DOI: 10.48550/ARXIV.2112.10752. URL: https://arxiv.org/abs/2112.10752.

[33] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[34] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 0028-0836. DOI: 10.1038/nature16961.

[35] Uriel Singer et al. *Make-A-Video: Text-to-Video Generation without Text-Video Data*. 2022. DOI: 10.48550/ARXIV.2209.14792. URL: https://arxiv.org/abs/2209.14792.

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

[37] Yuval Tassa et al. *DeepMind Control Suite*. 2018. DOI: 10.48550/ARXIV.1801.00690. URL: https://arxiv.org/abs/1801.00690.

[38] *The Otter*. URL: https://www.maritimerobotics.com/otter.

[39] Anete Vagale et al. "Path planning and collision avoidance for autonomous surface vehicles II: a comparative study of algorithms". In: *Journal of Marine Science and Technology* 26.4 (Feb. 2021), pp. 1307–1323. DOI: 10.1007/s00773-020-00790-x. URL: https://doi.org/10.1007/s00773-020-00790-x.

[40] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.7782 (Oct. 2019), pp. 350–354. DOI: `10.1038/s41586-019-1724-z`. URL: `https://doi.org/10.1038/s41586-019-1724-z`.

[41] Yara. *Yara Birkeland*. 2023. URL: `https://www.yara.com/news-and-media/media-library/press-kits/yara-birkeland-press-kit/`.