Masteroppgave

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

Ivan Håbjørg Kingman

Deep Reinforcement Learning Applied to Targeted Oceanographic Sampling for an Autonomous Underwater Vehicle

Comparing Machine Learning and Model Based Approaches in a Simulated Environment

Masteroppgave i Kybernetikk og robotikk
Veileder: Anastasios Lekkas
Medveileder: Andreas Våge
Juni 2021

NTNU
Kunnskap for en bedre verden

Ivan Håbjørg Kingman

Deep Reinforcement Learning Applied to Targeted Oceanographic Sampling for an Autonomous Underwater Vehicle

Comparing Machine Learning and Model Based Approaches in a Simulated Environment

Masteroppgave i Kybernetikk og robotikk
Veileder: Anastasios Lekkas
Medveileder: Andreas Våge
Juni 2021

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

NTNU

Kunnskap for en bedre verden

Abstract

Deep Reinforcement Learning (DRL) was applied in an attempt to enable an Autonomous Underwater Vehicle (AUV) to seek out hotspots of plankton in a simulated environment. Procedurally generated plankton data was used to provide a training environment for a dynamically modelled AUV, equipped with guidance and control systems. The learning agent was given a set of high level actions to choose from, and tasked with choosing actions to maximize encountered plankton while seeking out a patch of high plankton density, referred to as the plankton hotspot. The performance of the agent was compared to to a traditional pathfinding approach to the problem, namely the A* algorithm. The comparison revealed no clear benefit to the machine learning approach over the traditional model based approach, but indicated that targeted oceanographic sampling to some extent was achieved. Due to the highly simplified nature of the environment simulation, along with possibly insufficient training of the machine learning agent, the results are inconclusive. More work is needed to develop a more realistic simulation environment, specifically with real world plankton data, environment uncertainty, and ocean currents to simulate the dynamically varying biomass, defining a more complex problem where the machine learning approach may lend its powerful capability to targeted sampling in an uncertain and dynamic environment.

Sammendrag

Dyp forsterkende læring ble benyttet i et forsøk på å få en autonom undervannsdrone til å oppsøke biologiske varmepunkter ("hotspots") av plankton i et simulert miljø. Prosedyrisk generert planktondata ble benyttet for å danne et læringsmiljø for en dynamisk modell av en autonom undervannsdrone, utstyrt med styrings- og reguleringssystemer. Læreagenten ble presentert med et sett av handlinger av høy abstraksjonsgrad å velge fra, implementert som veipunkter for styringssystemet, og ble gitt i oppgave å velge handlinger for å maksimere plankton den måtte komme over mens den søkte etter et område med høy planktontetthet, omtalt som planktonvarmepunktet. Agentens ytelse ble sammenlignet med en tradisjonell stifinneralgoritme, nemlig A* algoritmen. Sammenligningen avdekte ingen tydelig fordel ved maskinlæringstilnærmingen over den tradisjonelle modellbaserte tilnærmingen, men indikerte at målrettet oseanografisk prøvetakning ble oppnådd til en viss grad. Ettersom miljøet er høyst oversimplifisert, samt potensielt utilstrekkelig trening av maskinlæringsagenten, er det vanskelig å trekke noen konkrete slutninger. Videre arbeid er nødvendig for å utvikle et mer realistisk simulert miljø, nærmere bestemt med planktondata fra den virkelige verden, usikkerhet i miljøet og strømninger i havet for å simulere den dynamisk varierende driften av biomasse i havet, og derved skape en mer kompleks problemstilling, hvor maskinlæringstilnærmingen kan gjøre nytte av dens mektige egenskaper for målrettet prøvetakning et usikkert og dynamisk miljø.

Acknowledgements

I would like to thank my supervisors Andreas Våge and Anastasios Lekkas for their guidance, expertise and encouragement. Their expertise and insight has taught me a lot about machine learning and control, and how to think with a scientific mindset. Next, I would like to thank my close friends Erik and Ida for always cheering me up during our many lunch breaks together. A special thank you to Erik, who first encouraged me to pursue a degree in cybernetics, and successfully tricked me into believing I could do it. I would also like to thank my mother, Anne, whose love and support has seen me through these challenging years. Thank you for always believing in me. And my sister, Maria, whose humor and cheerfulness inspires me to face adversity with a smile. A special thank you to my girlfriend Marte, whose insight and intellect has helped me solve many problems, and who has been there for me no matter what. Thank you for enduring my incoherent ramblings about reward functions, neural networks and plankton, and for being the highlight of my day.

Contents

Figures

Tables

Acronyms

6-DOF Six Degrees of Freedom. 3

AI Artificial Intelligence. 2, 15, 27

ALE Arcade Learning Environment. 27

ANN Artificial Neural Network. 22, 25, 26

AUV Autonomous Underwater Vehicle. i, ix, xvii, 1–3, 8–11, 33–40, 43–45, 48, 49, 53, 54, 57–59

CB Center of Buoyancy. 9

CG Center of Gravity. 7, 9

CNN Convolutional Neural Network. 26, 27

CO Coordinate Origin. 9, 13

DFFN Deep Feed Forward Network. 22, 24, 25

DL Deep Learning. 2, 3, 22, 27

DQN Deep Q-Network. ix–xii, xvii, 2, 3, 28, 30, 33, 35, 37, 38, 40, 41, 43, 44, 46–55, 57, 58, 65–90

DRL Deep Reinforcement Learning. i, 2, 3, 27, 33, 55, 57

GNC guidance navigation and control. 12

GP Gaussian Process. 15

GPR Gaussan Process Regression. 4, 40, 58

i.i.d. independent and identically distributed. 29

LOS line-of-sight. ix, 13, 14, 35, 37

MDP Markov Decision Process. 17–19, 33

ML Machine Learning. 2, 3, 35, 37, 54, 55, 57–59

MSE Mean Squared Error. 26, 28, 30

NED North-East-Down. 5, 7, 9, 13, 34

NTNU Norwegian University of Science and Technology. xvii

PID Proportional Derivative Integral. 11, 12, 37

ReLu rectified linear unit. 24

RL Reinforcement Learning. 2, 3, 15, 17–20, 22, 27, 29, 55

Preface

This document is my Master's Thesis for the degree of Cybernetics and Robotics at Norwegian University of Science and Technology (NTNU). The projet work spans the period from 4-th of January to the 7-th of June of 2021, and was supervised by associate professor Anastasios Lekkas and PhD candidate Andreas Våge, both at NTNU. The work in this thesis builds on my previous work done as part of the course TTK4550 at NTNU during the fall of 2020, commonly referred to as a thesis pre-project.

Originally, the context for this thesis was the AILARON project at NTNU. As my work continued, the goal of the project diverged somewhat from its original scope, but the motivation for the research question presented in this thesis should be understood with the AILARON project in mind.

Part of the project work presented in this paper consists of software, some of which is borrowed, some of which is developed by the author. An overview of the software contributions is given here. Additionally, the parts of the software and that is borrowed is indicated with comments in the source code.

The project makes use of a computer simulated Autonomous Underwater Vehicle (AUV). The source code simulating this model is mostly borrowed from the GitHub repository [1], with some minor modifications. This does not include the guidance module, which is developed by the author of this paper. Credit goes to the original author. Considerable parts of the plankton model generation is borrowed from one of the author's supervisors, Andreas Våge. The relevant parts is clearly indicated in the source code. The A* algorithm is taken from an implementation of A* found here [2], but has been modified to fit the specific application. Credit goes to the original author. The implementation of the DQN agent was done following a online tutorial. Although the implementation is made by the author of this thesis, it bears a resemblance to the original source code, available here [3].

Core parts of the source code makes use of third party Python libraries. Most notable is the use of Tensorflow and Keras to build, train, manage and use the neural networks used as part of the DQN agent. Additionally, the environment is developed following the OpenAI gym interface [4].

Chapter 1

Introduction

1.1 Background and Motivation

The oceans have served humankind throughout the ages, as a source of food, as means of transportation, providing natural resources, predicting the weather, and giving insight to the life of our planet as a whole. Our ability to understand and describe the ocean is key to many industrial and scientific endeavours alike.

Oceanography, the description of the ocean, relies on spatial samples of both physical and ecological phenomena. But the oceans are vast, and all areas of it are insufficiently sampled. This is known as the sampling problem of oceanography, and is in fact the largest source of error in our understanding of the ocean [5].

Traditionally, sampling techniques have relied on exhaustive grid-search methods carried out by personnel on ships. This is laborious and inefficient at best, and practically impossible at worst. Over the past decade, this task has been leveraged by the advent of increasingly robust and affordable mobile robotic platforms such as the Autonomous Underwater Vehicle (AUV), enabling autonomous collection of oceanographic data. The use of AUVs have been studied to detect the *thermocline* [6], [7], locating seafloor hydrothermal vents [8] and even tracing and surveying chemical plumes [9] and oil plumes [10]. The latter article was written in light of the Gulf of Mexico oil spill response of 2010.

Nevertheless, in order to optimize the sampling, the AUV should be equipped with the intelligence to know *where* to look, given its current surroundings. This is known as *targeted sampling*, allowing the sampling efforts to be concentrated on regions of high scientific interest. Algorithms for this type of sampling have been developed and proven successful in studying a range of oceanographic phenomena such as harmful algal blooms, coastal upwelling fronts and microbial processes in open-ocean eddies [11], and upwelling and internal waves on the west coast of Mid-Norway [12]. Targeted oceanographic sampling has also been studied to gather samples within the deep chlorophyll maximum layer to gain insight in microbial oceanography north of the island Maui, Hawaii [13].

Targeted sampling is ultimately a question of mapping observations in the

form of on-board sensor data to actions that are likely to realize some predefined sampling goal. The robot is essentially told how to plan. But could this behaviour be taught through Artificial Intelligence (AI) learning? The field of Machine Learning (ML) has in recent years witnessed the marrying of two previously separate approaches to the learning problem, namely Deep Learning (DL) and Reinforcement Learning (RL), giving birth to the field of Deep Reinforcement Learning (DRL). Although traditional RL has seen some success, e.g. optimizing quadrupedal trot gait for a specific robot [14] or inverted autonomous helicopter flight [15] amongst others, traditional RL methods lacks scalability and have been inherently limited to low-dimensional problems. The advent of DL has in recent years dramatically improved state of the art tasks such as language translation, object detection and speech recognition [16], due to its powerful ability to derive structure from high dimensional input data. Applying this capability to RL methods is currently enabling these methods to scale to previously intractable problems by freeing the RL approach from what is known as *the curse of dimensionality*.

The use of DRL to derive control policies has indeed proven successful in yielding interesting and impressive results. Kickstarting the interest for DRL in 2015, an algorithm capable of playing a range of Atari 2600 video games simply from observing the pixels of the game, and even beating the best human players, was developed [17]. In 2016, the first algorithm to successfully beat the world champion of Go was developed using DRL and tree search [18].

Since then, DRL has also been applied in robotics, enabling motion control policies to be taught directly from visual input. In [19] a robot was enabled to accomplish a range of manipulation tasks requiring close coordination between vision and control. A robot was able to successfully grasp novel object training on large amounts of data using RL in [20].

1.2 Research Question and Objective

The work presented in this thesis evaluates the application of DRL methods to learn and implement targeted sampling for an AUV in a simulated ocean environment. The ocean phenomenon in question is the density of plankton of the upper water column, and the desired behaviour is choosing actions, implemented as waypoint generation, to localize areas of high plankton density, and encountering as much plankton as possible along the way.

Two different approaches will be considered and compared to highlight the advantages and challenges of utilizing DRL for targeted sampling. The Deep Q-Network (DQN) approach will seek to *learn* pathfinding with no prior information of the goal state. This ML approach will be compared to a traditional pathfinding approach, namely the A* algorithm. As such, this thesis poses the following research questions:

- How is the performance of the DQN agent compared to pathfinding using

A* with regards to

- Reward returned by the environment
- Encountered plankton
- Ability to locate the hotspot

- What are the challenges of applying DRL to achieve targeted sampling behaviour?
- What are the potential advantages of applying DRL to achieve targeted sampling behaviour?

1.3 Contributions

Current work on targeted sampling has focused on model based approaches, and has done so successfully. The work presented in this project presents a novel approach to solving the sampling problem of oceanography. Albeit far from any complete algorithm or definitive answer, the results of this thesis may serve as a proof of concept for introducing ML, fuelled by its recent advances, a part of the solution.

The main results of this work is a simulated ocean environment consisting of a dynamically modelled AUV and a procedurally generated topological map of plankton density, implemented as an OpenAI gym environment. As such, different algorithms or learning agents may be trained and tested for different metrics within this environment, such as locating the hotspot, or maximizing the encountered plankton.

Additionally, two solutions to a specific problem within this environment are implemented. A modified A* search algorithm, and a DQN learning agent. These solutions are tested in the environment, and their performance analyzed and compared.

A dynamic model of the AUV was interfaced with the environment, simulating the AUV in Six Degrees of Freedom (6-DOF) with control input to propeller, rudder and elevator. A guidance and control system was developed on top of the AUV simulator, allowing high level abstract actions selected by the agent to be converted into waypoints and control inputs.

All source code with instructions to reproduce or build upon the work is available at [21].

1.4 Structure of the Report

chapter 2 presents the theoretical background material necessary to understand the material presented in subsequent chapters. The topics covered in section 2.1 cover the dynamic modelling of marine craft dynamics, based on the works of [22]. Both RL and DL are presented in section 2.5 and section 2.6 respectively to provide context and background for the DQN algorithm, presen-

ted in section 2.8. Additional topics discussed include Gaussan Process Regression (GPR) in section 2.4, the means by which synthetic plankton data was generated, along with classical control theory and guidance in section 2.2 and section 2.3 respectively. Finally, the A* algorithm is briefly presented in section 2.9.

The application of the topics presented in chapter 2 to achieve the questions raised in section 1.2 are presented in chapter 3 which describes the details of the ocean environment model, given in section 3.1, the methods used to interact with the environment, section 3.2, along with a description of the organization of the software implementing the agents and the environment in section 3.3. The performance of the agents in the ocean environment is presented and discussed in chapter 4. A brief summary, along with some remarks on future work is given in chapter 5.

Chapter 2

Background

2.1 Modelling Marine Craft Dynamics

In order to simulate the motions of a marine craft, a model of the vehicle is required. Such a model is given by the vehicles *dynamics*, divided into two parts, namely *kinematics* and *kinetics*. The former is the study of purely geometrical aspects of motion, whereas the latter includes an analysis of forces and moments causing the motion. The overarching goal of section 2.1 is to present the marine craft equations of motion, and show that they can be written as a set of matrix equations

$$\begin{aligned} \dot{\boldsymbol{\eta}} &= \boldsymbol{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu} \\ \boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) + \boldsymbol{g}_0 &= \boldsymbol{\tau} + \boldsymbol{\tau}_{\text{wind}} + \boldsymbol{\tau}_{\text{wave}} \end{aligned} \tag{2.1}$$

The concepts presented in section 2.1 is based on the material of [22]. The marine craft kinematics are presented in section 2.1.1, and the kinematics in section 2.1.1.

2.1.1 Kinematics

The purpose of this section is to arrive at the kinematic equation of eq. (2.1), that is

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu}. \tag{2.2}$$

This equation essentially gives the relationship between how a marine craft changes its position and what its velocities are. The generalized position $\boldsymbol{\eta}$ is given by

$$\boldsymbol{\eta} = [x^n, y^n, z^n, \phi, \theta, \psi]^\top. \tag{2.3}$$

These coordinates are specified with respect to the North-East-Down (NED) frame, denoted $\{n\}$ where the x-axis points towards the true north, i.e. the North Pole, y-axis points to the east, z-axis points down towards the center of the earth. The body-fixed velocity vector $\boldsymbol{\nu}$ is given by

$$\boldsymbol{\nu} = [u, v, w, p, q, r]^\top. \tag{2.4}$$

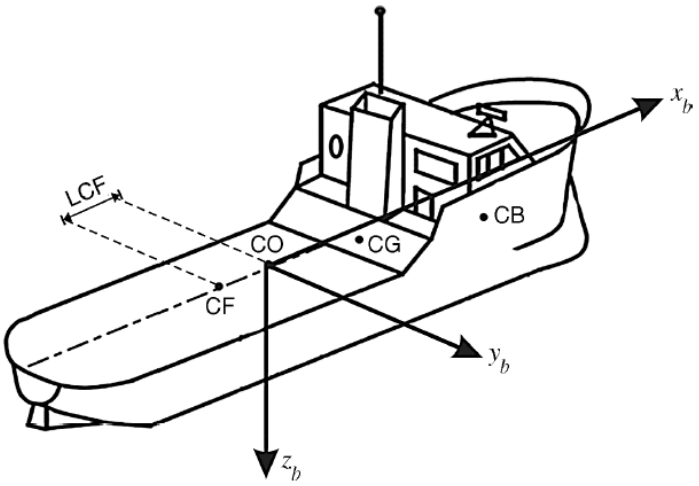

Figure 2.1: The body-fixed reference frame of a marine craft, along with points of interest within this frame. Figure courtesy of [22].

The body-fixed frame $\{b\}$ is rigidly attached to the marine craft, with the x-axis directed from the aft to the fore of the vessel, y directed starboard, and z directed top to bottom. Finally, the matrix $\boldsymbol{J}_\Theta(\boldsymbol{\eta})$ defines the coordinate transformation between $\dot{\boldsymbol{\eta}} = [x^n, y^n, z^n, \phi, \theta, \psi]^\top$ and $\boldsymbol{\nu}$ and is given by

$$\boldsymbol{J}_\Theta(\boldsymbol{\eta}) = \begin{bmatrix} \boldsymbol{R}(\boldsymbol{\Theta}_{nb}) & \boldsymbol{0}_{3\times3} \\ \boldsymbol{0}_{3\times3} & \boldsymbol{T}(\boldsymbol{\Theta}_{nb}) \end{bmatrix}. \tag{2.5}$$

where $\boldsymbol{R}_b^n$ is the rotation matrix between the frames $\{n\}$ and $\{b\}$ given by

$$\boldsymbol{R}_b^n = \begin{bmatrix} c\psi c\theta & -s\psi c\phi + c\psi s\theta s\phi & s\psi s\phi + c\psi c\phi s\theta \\ s\psi c\theta & c\psi c\phi + s\phi s\theta s\psi & -c\psi s\phi + s\theta s\psi c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi. \end{bmatrix} \tag{2.6}$$

and the matrix transformation $\boldsymbol{T}(\boldsymbol{\Theta}_{nb})$ is given by

$$\boldsymbol{T}(\boldsymbol{\Theta}_{nb}) = \begin{bmatrix} 1 & \mathrm{s}\phi\mathrm{t}\theta & \mathrm{c}\phi\mathrm{t}\theta \\ 0 & \mathrm{c}\phi & -\mathrm{s}\phi \\ 0 & \mathrm{s}\phi/\mathrm{c}\theta & \mathrm{c}\phi/\mathrm{c}\theta \end{bmatrix}. \tag{2.7}$$

First, the generalized coordinates for a marine craft, along with the frames of reference in which they are specified, are presented.

2.1.2 Rigid Body Kinetics

The purpose of this section is to give a brief description of the kinetic equations of eq. (2.1), that is

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) + \boldsymbol{g}_0 = \boldsymbol{\tau}. \tag{2.8}$$

Table 2.1: Conventional notation for marine vessels

DOF		Force	Velocity	Position and orientation
1	along x (surge)	X	u	x^n
2	along y (sway)	Y	v	y^n
3	along z (heave)	Z	w	z^n
4	about x (roll)	K	p	ϕ
5	about y (pitch)	M	q	θ
6	about z (yaw)	N	r	ψ

The term $\boldsymbol{\tau}$ on the right-hand side of eq. (2.8) was briefly mentioned in eq. (2.2) and is the vector of generalized forces in the NED-frame. eq. (2.8) thus gives the relationship between the forces applied to the marine craft, and how this changes the linear and angular acceleration of the craft. Together with eq. (2.2), this gives a complete description on how forces changes the crafts position and orientation, giving a model of the dynamics of the marine craft.

System Inertia Matrix

The matrix $\boldsymbol{M}$ is known as the system *system inertia matrix*, and is given by both *rigid body inertia matrix* $\boldsymbol{M}_{RB}$ and the *added mass inertia matrix*, that is

$$\boldsymbol{M} = \boldsymbol{M}_{RB} + \boldsymbol{M}_A. \tag{2.9}$$

Conceptually, inertia is a rigid body resisting change to its velocity. For a marine craft, this resistance to change is caused by the physical properties of the craft as a rigid body, given by $\boldsymbol{M}_{RB}$ and the fact that moving a craft in the water also requires moving some water with it, resulting in *added mass* given by $\boldsymbol{M}_A$

The system rigid body inertia matrix $\boldsymbol{M}_{RB}$ is given by

$$\boldsymbol{M}_{RB} = \begin{bmatrix} m\boldsymbol{I}_{3\times 3} & -m\boldsymbol{S}\left(\boldsymbol{r}_g^b\right) \\ m\boldsymbol{S}\left(\boldsymbol{r}_g^b\right) & \boldsymbol{I}_b \end{bmatrix} \tag{2.10}$$

where $\boldsymbol{I}_{3\times 3}$ is the 3×3-identity matrix, m is the mass of the marine craft, $\boldsymbol{S}$ is the skew-symmetric matrix used as a cross-product operator according to **??**, and $\boldsymbol{r}_g^b$ is the CG given in the body frame. The term $\boldsymbol{I}_b$ is given by $\boldsymbol{I}_g - m\boldsymbol{S}^2(\boldsymbol{r}_g^b)$ where $\boldsymbol{I}_g$ is the *inertia matrix* about CG, defined as

$$I_g := \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{yx} & I_y & -I_{yz} \\ -I_{zx} & -I_{zy} & I_z \end{bmatrix}, \quad I_g = I_g^\top > 0. \tag{2.11}$$

The diagonal terms of eq. (2.11) are the moments of inertia about the vehicles x, y and z axis. The off-diagonal terms are the products of inertia.

It can be shown [22] that the system inertia matrix $\boldsymbol{M}_A$ for an AUV is given by

$$\boldsymbol{M}_A = -\operatorname{diag}\left\{X_{\dot{u}}, Y_{\dot{v}}, Z_{\dot{w}}, K_{\dot{p}}, M_{\dot{q}}, N_{\dot{r}}\right\} \tag{2.12}$$

under the assumption that the vehicle operates at low speeds, is completely submerged and is symmetric about three planes. The coefficients of eq. (2.12) are known as *hydrodynamic added mass derivatives* and are found either by a hydrodynamic program, or experimentally from observing the vehicle dynamics.

Coriolis-Centripetal Matrix

The system coriolis-centripetal matrix $\boldsymbol{C}$ describe forces on the marine craft resulting from the fact that the craft rotates within the inertial frame. As with the system inertia matrix, the coriolis-centripetal matrix is a combination of rigid body, $\boldsymbol{C}_{RB}$, and added mass $\boldsymbol{C}_A$ properties, i.e.

$$\boldsymbol{C} = \boldsymbol{C}_{RB} + \boldsymbol{C}_A. \tag{2.13}$$

It can be shown that this matrix can be obtained directly from the system inertia matrix [22]. This is given by

$$\boldsymbol{C}(\boldsymbol{\nu}) = \begin{bmatrix} \mathbf{0}_{3\times 3} & -\boldsymbol{S}(\boldsymbol{M}_{11}\boldsymbol{\nu}_1 + \boldsymbol{M}_{12}\boldsymbol{\nu}_2) \\ -\boldsymbol{S}(\boldsymbol{M}_{11}\boldsymbol{\nu}_1 + \boldsymbol{M}_{12}\boldsymbol{\nu}_2) & -\boldsymbol{S}(\boldsymbol{M}_{21}\boldsymbol{\nu}_1 + \boldsymbol{M}_{22}\boldsymbol{\nu}_2) \end{bmatrix} \tag{2.14}$$

where $\boldsymbol{\nu}_1 = [u, v, w]^\top$ and $\boldsymbol{\nu}_2 = [p, q, r]^\top$. The matrices $\boldsymbol{M}_{11}, \boldsymbol{M}_{12}, \boldsymbol{M}_{21}, \boldsymbol{M}_{22}$ are the sub-matrices of the system inertia matrix.

Hydrodynamic Damping Matrix

Hydrodynamic damping is caused by the water resisting the relative velocity of the marine craft. This happens as a result of several different phenomena, like skin friction or damping caused by the vortexes the craft creates in the water.

Instead of describing individual matrices for each phenomenon, the damping matrix may be separated into linear and non-linear damping system matrices, given by

$$\boldsymbol{D}(\boldsymbol{\nu}_r) := \boldsymbol{D} + \boldsymbol{D}_n(\boldsymbol{\nu}_r). \tag{2.15}$$

The linear damping system matrix $\boldsymbol{D}$ for an AUV is given by

$$\boldsymbol{D} = -\begin{bmatrix} X_u & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_v & 0 & Y_p & 0 & Y_r \\ 0 & 0 & Z_w & 0 & Z_q & 0 \\ 0 & K_v & 0 & K_p & 0 & K_r \\ 0 & 0 & M_w & 0 & M_q & 0 \\ 0 & N_v & 0 & N_p & 0 & N_r. \end{bmatrix} \tag{2.16}$$

The non-linear damping system matrix is given by

$$\boldsymbol{D}_n(\boldsymbol{\nu}_r) = -\begin{bmatrix} X_{|u|u}|u_r| & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_{|v|v}|v_r| + Y_{|r|v}|r| & 0 & 0 & 0 & Y_{|v|r}|v_r| + Y_{|r|r}|r| \\ 0 & 0 & Z_{|w|w}|w_r| & 0 & 0 & 0 \\ 0 & 0 & 0 & K_{|p|p}|p| & 0 & 0 \\ 0 & 0 & 0 & 0 & M_{|q|q}|q| & 0 \\ 0 & N_{|v|v}|v_r| + N_{|r|v}|r| & 0 & 0 & 0 & N_{|v|r}|v_r| + N_{|r|r}|r| \end{bmatrix}.$$

The constants of these matrices are referred to as *hydrodynamic linear damping coefficients*, and are either found through hydrodynamic programs, or experimentally.

Hydrostatic Forces

In hydrostatic terminology, the forces of gravity and buoyancy are known as *restoring forces*. The dynamics of these restoring forces are given in the vector $\boldsymbol{g}(\boldsymbol{\eta})$ which is given by

$$\boldsymbol{g}(\boldsymbol{\eta}) = \begin{bmatrix} (W-B)\cdot \mathrm{s}\theta \\ -(W-B)\cdot \mathrm{c}\theta \cdot \mathrm{s}\phi \\ -(W-B)\cdot \mathrm{c}\theta \cdot \mathrm{c}\phi \\ -\left(y_g W - y_b B\right)\cdot \mathrm{c}\theta \cdot \mathrm{c}\phi + \left(z_g W - z_b B\right)\cdot \mathrm{c}\theta \cdot \mathrm{s}\phi \\ \left(z_g W - z_b B\right)\cdot \mathrm{s}\theta \quad + \left(x_g W - x_b B\right)\cdot \mathrm{c}\theta \cdot \mathrm{c}\phi \\ -\left(x_g W - x_b B\right)\cdot \mathrm{c}\theta \cdot \mathrm{s}\phi - \left(y_g W - y_b B\right)\cdot \mathrm{s}\theta \end{bmatrix}. \tag{2.17}$$

under the assumption that the entire rigid body is submerged. For a rigid body submerged in water, the gravitational force $\boldsymbol{f}_g^b$ acts on the CG defined by $\boldsymbol{r}_g^b := \left[x_g, y_g, z_g\right]^\top$ with respect to CO. Similarly, the force of buoyancy $\boldsymbol{f}_b^b$, acts through the CB, defined by $\boldsymbol{r}_b^b := [x_b, y_b, z_b]^\top$. Since these forces only act in the vertical plane, they are given in the NED frame as

$$f_g^n = \begin{bmatrix} 0 \\ 0 \\ W \end{bmatrix} \quad \text{and} \quad f_b^n = -\begin{bmatrix} 0 \\ 0 \\ B \end{bmatrix} \tag{2.18}$$

where $W = mg$ and $B = \rho g \nabla$. Here, m denotes the mass of the vehicle, ∇ the volume of fluid displaced by the vehicle, ρ is the density of the fluid, and g is the acceleration of gravity, positive downwards. This is shown in fig. 2.2.

The term $\boldsymbol{g}_0$ of eq. (2.8) is all zero for AUVs, as it describes the hydrostatic forces and moments from *ballast systems*, which are not relevant for this application.

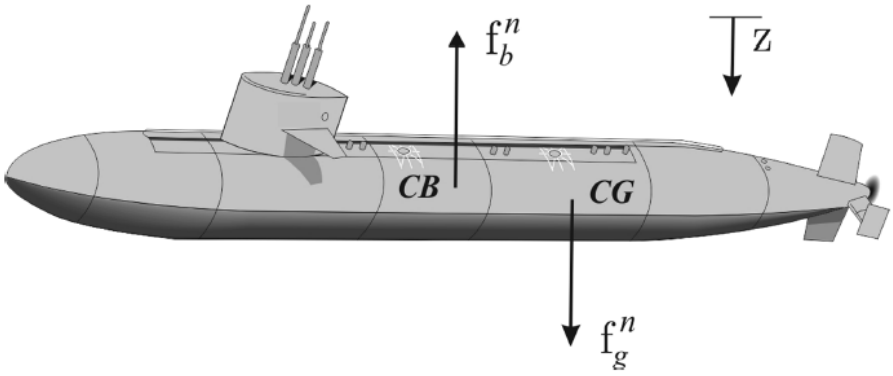

Figure 2.2: The restoring forces on a submerged marine craft. Figure courtesy of [22].

Table 2.2: Actuators and control variables.

Actuator	Control Input
Main propeller	rpm
Aft rudder	angle
Elevator	angle

2.1.3 Kinetics and Kinematics

Combining eq. (2.2) and eq. (2.8) gives the dynamic equations for a marine-craft, with the system matrices specified for an AUV, repeated here for convenience

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu}$$
$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) + \boldsymbol{g}_0 = \boldsymbol{\tau} + \boldsymbol{\tau}_{\text{wind}} + \boldsymbol{\tau}_{\text{wave}}$$

2.1.4 Control Allocation

The generalized force vector $\boldsymbol{\tau}$ of eq. (2.1) is the means by which the marine craft may be steered. These forces are generated by the *actuators* of the marine craft. For an AUV, actuators include main propeller, capable of applying a force F_x along the x-axis of the body frame, an aft rudder, which may be deflected to induce moments about the vehicles z-axis, and elevators, which may be deflected to induce moments about the vehicles y-axis. In this case, the degrees of freedom outnumber the actuators, leading to an *underactuated system*.

The input to the actuators is not given in terms of the force the actuators should produce. In fact, input is typically given as a voltage. Assuming a system for supplying the appropriate voltage given a specified actuator reference exists, the input to the different actuators are shown in table 2.2.

Let the control inputs to the actuators be given by $\boldsymbol{u} = [u_1, u_2, u_3]^\top$, where u_1 is the main propeller rpm, u_2 is the rudder deflection angle, and u_3 is the

elevator deflection angle. The resulting generalized force vector $\boldsymbol{\tau}$ is given by the matrix equation

$$\boldsymbol{\tau} = \boldsymbol{B}(\boldsymbol{\eta})\boldsymbol{u} \tag{2.19}$$

where $\boldsymbol{B}$ is a 6×3 matrix referred to as the *input matrix*, generally dependent on the AUV state $\boldsymbol{\eta}$. This dependency between the state and the control input is known as feedback control, and is discussed in the following section.

2.2 Control Theory

Control theory is the study of *control* or *regulation*. The purpose of applied control is to generate some automatic influence over a technical system or process to achieve some goal result. The purpose of this section is to introduce the Proportional Derivative Integral (PID)-controller and to provide definitions for terms and concepts within control theory. The material in this section is based on [23].

2.2.1 Control System and Process

A *control system*, or simply just *system*, is a collection components mutually affecting each other. Control theory is primarily concerned with *dynamic systems*, that is, systems where internal states change with time as a result of the components interacting. The component within a system subject to control is known as the *process*. For example, an AUV in the water is a process, whereas an AUV with navigation and autopilot is a system. The process is defined by a series of signals. Its *state* is a collection of attributes, generally time dependent, describing the configuration of a system. For an AUV the state is typically its position, orientation and their derivatives. The state of a system is changed through *control input*, the means by which a process is changed, more precisely, its state moved to some desired value. In the case of an AUV, control input is typically motor thrust and fin deflections, by which position and orientation may be changed. The state is generally considered internal to the process, meaning it is not necessarily known to other components in the control system. A *measurement* of a process is some function of the state giving insight to the state. A measurement may be considered the output of the process, the control signal an input.

2.2.2 Feedback Control

Control theory is not only the description of processes, but the design of *controllers* to generate control input to the system in some meaningful way. The purpose of a controller is to decide a control input that changes the system state to a certain value, known as the *setpoint* or *reference*. A variety of tools and methods exist for designing controllers. A controller design that has proven

powerful in many industrial applications is the *feedback controller*. The key idea behind feedback control is to compute the control input as a function of the process output, or measurement. Viewing the controller as as a component within the control system, the input to this component is the output of the process, and the input to the process is in turn the output of the controller. This creates a *feedback loop*, giving rise to the name of this control design scheme.

The way the control input is computed from the process output is by introducing an error variable

$$\boldsymbol{e} = \boldsymbol{x}_d - \boldsymbol{x} \tag{2.20}$$

giving the distance between the desired and current state. Rather intuitively, the control input is proportional to the error: The further away the process is from the desired state, the more control input is needed. Additionally the time derivative and integral of the error is included in the control law, giving rise to the Proportional Derivative Integral (PID) controller:

$$\boldsymbol{u}(t) = \boldsymbol{K}_p \boldsymbol{e}(t) + \int \boldsymbol{K}_i \boldsymbol{e}(t) dt + \boldsymbol{K}_p \boldsymbol{e}(t) \frac{d}{dt} \tag{2.21}$$

where $\boldsymbol{K}_p, \boldsymbol{K}_i, \boldsymbol{K}_d$ are known as proportional, integral and derivative gains respectively. The motivation for including a derivative and integral term is to provide *damping* and to deal with constant *stationary offsets*. The derivative term will limit the use of control input as the error is rapidly decreasing or generate more control input even though the error is small, if the error is rapidly increasing. The integral term will increase the control input if an error persists over time, handling cases where the control input from the proportional term is insufficient.

2.3 Guidance

The previous section introduced the notion of control. Control applied to vehicles is known as *motion control systems*, and are typically subsystems in a higher level system known as guidance navigation and control (GNC) systems. This section presents another component of GNC-systems, namely the *guidance* system. The purpose of a guidance system is to decide appropriate set-points to the motion control system. Its inputs are *waypoints*, locations in space the vehicle should reach. The third component in a GNC system is the *navigation system*. The purpose of the navigation system is to provide estimates of the vehicles position from available sensor data. These position estimates are in turn provided to both motion control and guidance systems, as these systems rely on position estimates to carry out their tasks. The purpose of this section is to describe the line of sight guidance, as presented in [22].

2.3.1 Path Following for Straight-Line Paths

In cases where the time at which waypoints are reached is arbitrary, meaning there is no temporal constraints imposed on the guidance system, waypoints can be reached by implementing *path following*. A target path may be generated as a straight-line segment between two consecutive waypoints. A frequently used method for path following is by *line-of-sight (LOS)* guidance: A LOS-vector is generated as a straight line segment between the craft and the target waypoint given in an inertial frame (for instance NED). The LOS-vector and the path objective can then be used to compute a desired heading, which in turn is fed to a heading control system. Additionally, surge velocity references are computed and provided to a separate velocity control system to ensure the waypoint is actually reached. For underwater vehicles, a third motion control system is required for depth control.

Keeping in mind that marine crafts may be subject to ocean currents, it is in practise the *course* and *speed* of the craft that is subject to control, rather than the heading and surge velocity.

Course Control

The desired course control reference χ_d for LOS-guidance can be implemented as

$$\chi_d(e) = \chi_\mathrm{p} + \chi_\mathrm{r}(e) \tag{2.22}$$

where $\chi_p = \alpha_k$ is the *path-tangential angle*, that is the angle from the NED x-axis pointing north to the line defining the target path. In the case that the path is defined as the line segment from the vehicle to the next waypoint, this course reference angle is sufficient. In the case that the vehicle is not on the path, this angle alone may steer the vehicle away from the waypoint. To ensure that the velocity of the craft is directed to a point on the path, the term $\chi_r(e)$ is necessary. It is referred to as the *velocity-path relative angle* and defined as

$$\chi_\mathrm{r}(e) := \arctan\left(\frac{-e}{\Delta}\right). \tag{2.23}$$

e is known as the *cross-track error* and is defined as the distance from the craft perpendicular to the path and can be obtained by

$$e(t) = -[x(t) - x_\mathrm{k}]\sin(\alpha_\mathrm{k}) + [y(t) - y_\mathrm{k}]\cos(\alpha_\mathrm{k}) \tag{2.24}$$

where x_k, y_k is the NED position of the previous way-point. The variable $\Delta(t)$ is known as the *lookahead-distance*. A circle of radius R, known as *circle of acceptance* is drawn around the crafts CO. The circle intersects the target path at two points, the latter of which is defined as x_{los}, y_{los}. fig. 2.3 illustrates the LOS-guidance angles in eq. (2.22), eq. (2.23), and eq. (2.24). The lookahead-distance is the distance from the craft position projected onto the target path along e, to x_{los}, y_{los} and can be found by

$$\Delta(t) = \sqrt{R^2 - e(t)^2}. \tag{2.25}$$

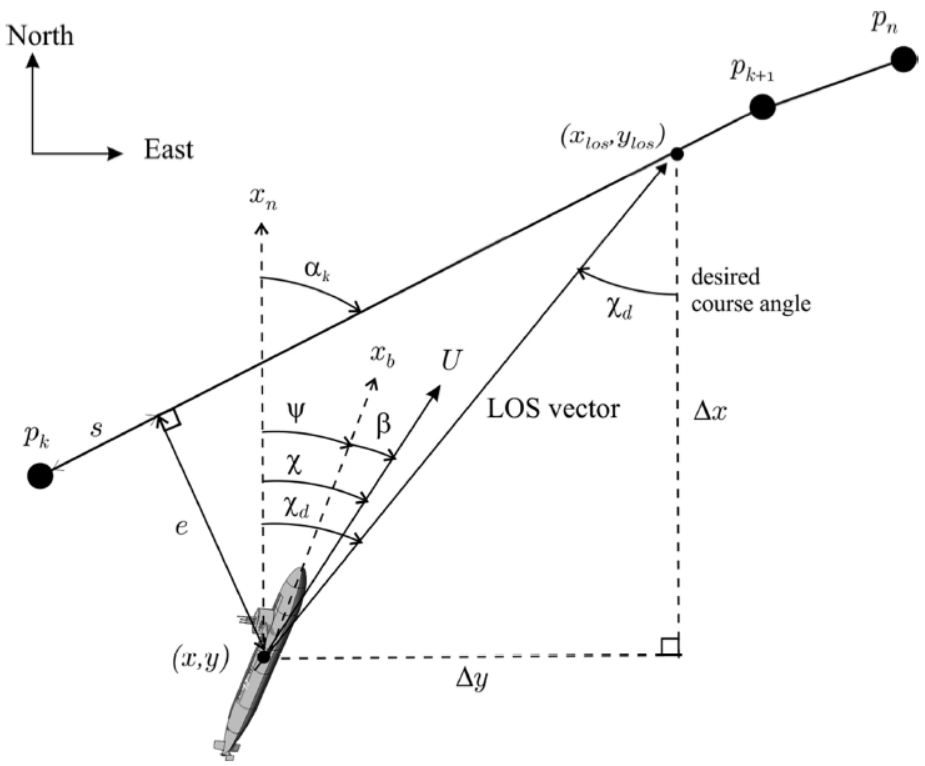

Figure 2.3: Definition of constants used in LOS guidance. Figure courtesy of [22].

As marine crafts typically are equipped with heading control systems rather than course control systems, the heading reference needs to be calculated from the course reference. This is done by subtracting the *crab angle* β, which indeed is the discrepancy between course and heading. The crab angle is defined by

$$\beta = \arcsin\left(\frac{v}{U}\right). \tag{2.26}$$

The crab angle is illustrated in fig. 2.3. The relation between the lookahead distance, the radius of the circle of acceptance and the cross-track-error given in eq. (2.25) is illustrated in fig. 2.4.

Speed Control

Assuming the vehicle is on the right course, meaning it is in some sense headed towards the next waypoint, the surge controller should ensure that the vehicle moves forward. As mentioned previously, the vehicle is generally subject to ocean currents, meaning the surge velocity is not necessarily equivalent with the rate at which the vehicle moves in the direction of the course vector.

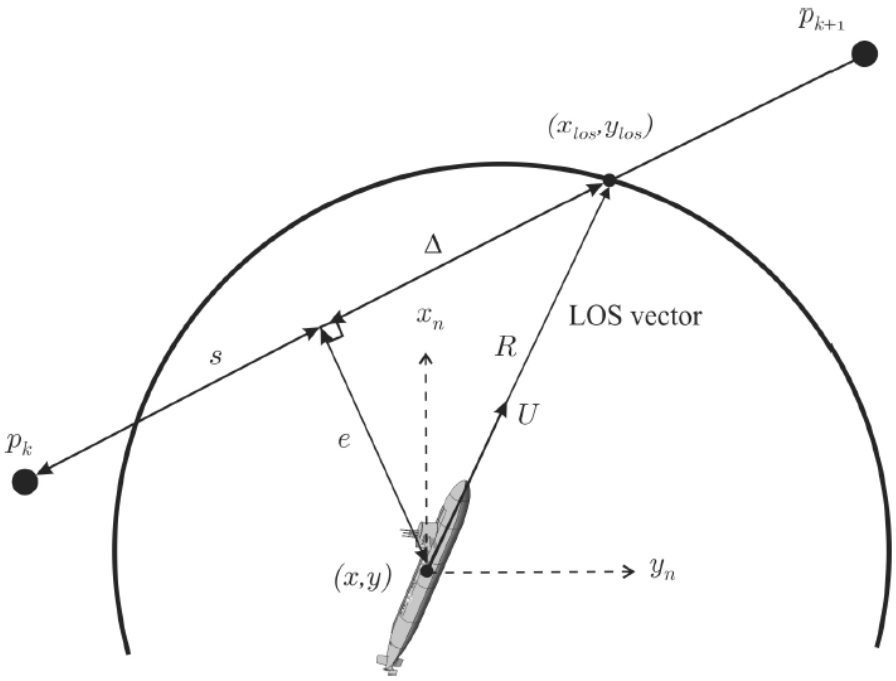

Figure 2.4: A geometric illustration of cross-track-error and lookahead distance. Figure courtesy of [22].

2.4 Gaussian Processes Model

A Gaussian Process (GP) is a probability distribution over possible functions that fit a set of points [24] and is essentially a collection of random variables with a multivariate normal probability density function. Domains where variables are allocated to spatial locations and depend on adjacent spatial locations, such as environmental heat-maps, are for this reason suitable to be modelled as a GP, as it allows the dependency to be modelled using covariance functions. Due to its representational flexibility, modelling by GP is a popular way of describing environmental processes [25]. GP may then be used as a regression task, predicting spatial data using prior knowledge, while also providing uncertainty measures for said estimates. This has been applied successfully in [12] and is discussed in [26].

2.5 Reinforcement Learning

Reinforcement Learning (RL) is at its core learning by doing. It is simultaneously a problem description, a class of solutions to said problem, and the name of the field studying the two [27]. RL falls within the broader field of machine learning, which in turn is a central problem to Artificial Intelligence (AI). The material presented in section 2.5 based in its entirety on [27].

2.5.1 Agent-Environment Framework

The framework for all reinforcement learning is the agent-environment framework. Some entity, referred to as an *agent* finds itself in an *environment*, which it can fully or partially observe. The goal of the agent is then to behave in some optimal way, without being told what this behaviour is. Instead, this behaviour is defined implicitly by feedback from the environment through a *reward signal*. This signal may be a function of what the agent did, its *actions* or where in the environment the agent is, its *state*, or both. Then, by remembering what actions and states led to high rewards, the agent is encouraged to repeat certain actions in certain states, thus reinforcing the optimal behaviour through learning.

This description, although intuitive, is somewhat informal. A state s can be thought of as a description of an instance of the environment. This description is made using a collection of relevant attributes of the environment.

For a mobile robot positioned on a square 2D surface, for example, a set relevant attributes would typically be its x- and y-position in some coordinate frame attached to the surface. Note here that the position of the robot itself is considered part of the environment, not the agent. The set of possible environment states, known as the *state space* is denoted S.

An action is denoted by a, and the set of actions by A. Actions are the agents means of changing the state. Continuing with the mobile robot example, one might imagine actions like moving up, down, right or left, or staying in place, and thereby changing the robots coordinates on the surface. Generally, potential actions are functions of the state, meaning only a subset of A may be available in a given state, giving rise to the notation $A(s) \subseteq A$. In the robots case, one might imagine a world where the robot cannot exit the 2D surface, making certain actions unavailable at the edges of the square. Although the agent may change the state through its actions, the state may generally change on its own, not as a consequence of the agents actions. One might imagine random gusts of wind, moving the robot around.

The reward signal is a scalar denoted by r and is as mentioned above a function of the state and action of the environment and agent. The reward signal determines what is to be considered good behaviour. If the goal of the robot is to stay on the surface and resist the gusts of wind, for example, a reasonable reward signal would be to associate higher rewards to coordinates closer to the centre of the square. This highlights the fact that even though the reward signal may be a function of the environment, it is indeed chosen by design to implicitly define what the purpose of the agent is.

So far, the notion of time has been overlooked. But the agent-environment setting is generally a dynamic process repeating in a simple loop. First, the agent finds itself in the initial state, denoted s_0. It then performs some action, a_0 and experiences a reward r_0, while transitioning to the next state s_1. This gives rise to the subscript t to indicate what time step the state, action, reward

and resulting state occurred. A collection of these four will be referred to as an *experience*, denoted e_t:

$$e_t \doteq \{s_t, a_t, r_t, s_{t+1}\} \tag{2.27}$$

This *agent-environment-loop* is illustrated in fig. 2.5

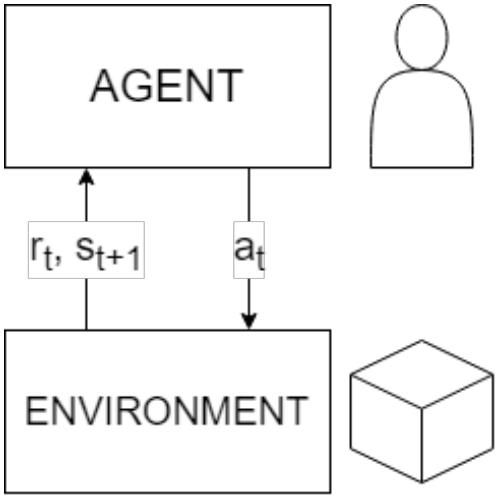

Figure 2.5: The agent-environment framework.

2.5.2 Markov Decision Process

A Markov Decision Process (MDP) provides a formalization of the concepts introduced in section 2.5.1. An MDP can be considered as a mathematically idealized form of the RL problem. This section defines the MDP structure in the context of the agent-environment framework described in section 2.5.1.

An MDP is formally defined as a four-tuple and a constant $\gamma \in [0, 1]$ known as the *discount factor*. The tuple consists of the action-space A, state-space S, as described in section 2.5.1, in addition to a reward function, and a state transition map. The reward function has indeed been mentioned previously, but is now formally defined as a mapping from the domain $S \times A \times S$ to that of real numbers $\mathbb{R}$ given by

$$R : S \times A \times S \to \mathbb{R} \tag{2.28}$$

or equivalently

$$R(s, a, s\prime) = r \tag{2.29}$$

where $s, s\prime \in S$, $a \in A$, and $r \in \mathbb{R}$. Note that the subscript t is omitted, and the consecutive state s_{t+1} is replaced by the superscripted state $s\prime$. As evident by eq. (2.29) the reward function is not only dependent on the resulting state and action taken, but also the current state s.

The state transition map describes how different actions affect the state. It can be thought of as the dynamics of the environment. These dynamics are

given by a stochastic process, highlighting the fact that transitions between states are subject to uncertainty and disturbances. The state transition map is formally defined as a mapping from the domain of a two states, s and $s\prime \in S$, and an action $a \in A$, to a probability $p \in [0,1]$, essentially giving the probability that taking action a in state s results in state $s\prime$, that is

$$T(s\prime \mid a, s) = p \tag{2.30}$$

Knowledge of eq. (2.30) and eq. (2.29) assumes some knowledge or model of the environment. Much of the merit of RL is that this knowledge is not necessarily required a priori, but learned.

Before describing the aforementioned discount factor γ and thereby completing the definition of an MDP, some details concerning the nature of the tasks carried out by RL agents are discussed. As briefly mentioned in section 2.5.1, the agent-environment-loop is a process subject to time, as actions, rewards and resulting states constitutes a chronological process. This raises the question of the duration of this process. This is of course dependent on the individual RL task: Some tasks may be a "one-shot" decision, such as classification of an image, others may have a finite or possibly set duration, whereas some may be considered complete once either of a certain set of states are reached, as is the case with e.g. chess. Tasks may even theoretically continue infinitely, for example classic arcade games with high-scores. This gives rise to different classes of environments, namely episodic, infinite and one-shot tasks. Episodic tasks are considered to be done under certain criteria, either when the duration has exceeded some limit, or when the environment state is a *terminal* state. Returning to the chess example, if either of the players kings are surrounded, the game is over, defining the configuration of pieces on the board as a terminal state. One typically plays multiple games of chess. In RL terminology, a single game would be referred to as an *episode*, that is the collection of experiences from $t = 0$ until the terminal state.

The reward function gives an immediate reward, and provides a way for the learning agent to learn the optimal behaviour. This is done by maximizing the future accumulated reward of the task, known as *return* G. The return at time t is the sum of future rewards, given by

$$G_t \doteq r_{t+1} + r_{t+1} + \ldots + r_T \tag{2.31}$$

where T is the final time step, possibly as a result of a terminal state. Maximizing eq. (2.31) thus requires some form of evaluation of potential future rewards. This is possible for episodic tasks. However, for potentially infinite tasks this may be problematic. This is leveraged by the discount factor $\gamma \in [0,1]$. Conceptually, the discount factor imposes a diminishing return on future rewards. The expected discounted return is given by

$$G_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.32}$$

This concludes the definition of an MDP.

2.5.3 Value and Policy

This section presents the concepts of value functions and policies. These concepts are not consider part of the MDP, but rather a toolbox to apply RL on the MDP framework. The most important result of this section is the *Bellman optimality equation for the action-value equation*, enabling many RL agents to find the best action to take in any state.

Although the MDP provides the framework for RL, it does not provide the solution. As mentioned in section 2.5.2, the purpose of the agent is to maximize eq. (2.32). Instinctively, one would think that this could be achieved by always selecting the action yielding the highest expected reward, either by using knowledge of the model if eq. (2.29) and eq. (2.30) are available, or using previous experiences as a basis for reasoning. This is known as a greedy approach, and will in some cases produce the desired behaviour. But in many cases, never thinking more than one step ahead is a poor strategy. Returning once again to chess, capturing a pawn might give a temporary lead, but may cost the game if the move exposes the queen. For this reason, chess scores during games are in fact not only determined by captured pieces, but the potential for winning the game, given the state of the board. This introduces the notion of *value* of a state. The value of a state is conceptually an indication of future reward one might expect from that state. But future rewards are not only dependent on the current state, but also every future action. For this reason, the value of a state is specified under some rule of what actions to take, known as a *policy*.

A policy can be thought of as a lookup table, specifying an action to take in a given state, ultimately dictating the behaviour of the agent. In fact, a policy is in many RL algorithms the end result, and serves as the means to maximizing eq. (2.32). Generally, a policy π is a mapping from a state to a probability distribution over actions, or equivalently, a mapping from an action and state to a probability $p \in [0, 1]$

$$\pi(a \mid s) \doteq p \tag{2.33}$$

for every $s \in S$ and every $a \in A(s)$. In the special case of a deterministic policy, the same action is always taken in a given state, somewhat simplifying the concept

$$\pi : S \to A. \tag{2.34}$$

The reason for having stochastic policies are discussed in section 2.5.4.

Returning to the concept of value, which is defined under a policy, the formal definition of the value V of a state s when following policy π is defined as

$$V_\pi(s) \doteq \mathbb{E}[G_t \mid s_t = s] \tag{2.35}$$

for all $s \in S$. With discounted rewards, inserting eq. (2.32) for G_t, eq. (2.35) becomes

$$V_\pi(s) \doteq \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]. \tag{2.36}$$

The function in eq. (2.35) is referred to as the *state-value function for policy* π. It gives the expected return when starting in state s and then following policy π. A similar concept is that of the *action-value function*. Whereas eq. (2.36) gives the value of a certain state, the action-value function also specifically gives the value of taking an action in a state, i.e. the expected return when starting in state s, taking action a and then following policy π. The action-value function is given by

$$Q_\pi \doteq \mathbb{E}[G_t \mid s_t = s, a_t = a] \tag{2.37}$$

for all $s \in S$ and $a \in A(s)$. Once again substituting eq. (2.32) into eq. (2.37), the action-value function is expressed as

$$Q_\pi \doteq \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right] \tag{2.38}$$

giving the expected return when starting in state s, taking action a, and then following policy π. These functions will collectively be referred to as *value functions*.

Although the policy may be the purpose of many RL algorithms, a value function is typically found first, and from that, a policy is derived. Calculating eq. (2.36) and eq. (2.38) is done using a fundamental property of these equations, known as the *Bellman Equation*. For eq. (2.38) this can be shown [27] to be given by

$$V_\pi(s) \doteq \sum_a \pi(a \mid s) \sum_{s'} T\left(s' \mid s, a\right)\left[R(s, a, s') + \gamma V_\pi\left(s'\right)\right]. \tag{2.39}$$

eq. (2.39) expresses the state-value function as a function of possible values of the next state, weighted by the probability that the current policy leads to that state.

Value functions are means of obtaining a policy, and a policy is ultimately a way of ensuring that the return is maximized. This introduces the need for a "best" policy, known as the *optimal policy*. A policy π is defined to be greater or equal to a policy π', denoted $\pi \geq \pi'$ if the expected return of π is greater or equal to that of π' for all states $s \in S$, more precisely $\pi \geq \pi' \iff V_\pi(s) \geq V_{\pi'}(s)$. A policy that is greater or as good as every other policy is known as an *optimal policy* and denoted π^*. An optimal policy may not necessarily be unique. The value functions under the optimal policy are known as the *optimal state-vale* and *optimal action-value functions*, respectively denoted and defined as

$$V^*(s) \doteq \max_\pi V_\pi(s) \tag{2.40}$$

for all $s \in S$, and

$$Q^*(s, a) \doteq \max_\pi Q_\pi(s, a) \tag{2.41}$$

for all $s \in S$ and $a \in A(s)$.

As previously mentioned, the Bellman equation provides a way of computing the value functions. The Bellman equation for the optimal value functions is known as *Bellman optimality equation*. As the optimal value function is under that of the optimal policy, it is intuitively clear that the Bellman optimality equation should be equal to the expected return when taking the *best action* of that state. The Bellman optimality equation for the state-value function can be shown [27] to be given by

$$V^*(s) \doteq \max_{a \in A(s)} \sum_{s' \in S} T\left(s,a,s'\right)\left(R\left(s,a,s'\right) + \gamma V^*\left(s'\right)\right). \tag{2.42}$$

Finally, the Bellman optimality equation for the action-value function is given by

$$Q^*(s,a) = \sum_{s' \in S} T\left(s,a,s'\right)\left(R\left(s,a,s'\right) + \gamma \max_{a'} Q^*\left(s',a'\right)\right). \tag{2.43}$$

Just like s' denotes the state at the next time step, so does a' denote the action at the next time step. The derivation from eq. (2.41) to eq. (2.43) can also be found in [27].

Computing eq. (2.43) may be done iteratively. This is readily done by turning the Bellman Optimality Equation to an iterative update rule

$$Q^*_{i+1}(s,a) = \sum_{s' \in S} T\left(s,a,s'\right)\left(R\left(s,a,s'\right) + \gamma \max_{a'} Q^*_i\left(s',a'\right)\right). \tag{2.44}$$

where i denotes the i-th iteration. Once the change between consecutive computations is sufficiently small, the value iteration is said to have converged to the true value.

Once eq. (2.42) and eq. (2.43) are given, the optimal policy is readily available. For the state-value function, this is given by

$$\pi^*(s) = \text{argmax}_a \sum_{s' \in S} T\left(s,a,s'\right)\left(R\left(s,a,s'\right) + \gamma V^*\left(s'\right)\right), \tag{2.45}$$

for all $s \in S$. The action-value function case is much simpler and given by

$$\pi^*(s) = \text{argmax}_a \, Q^*(s,a) \tag{2.46}$$

for all $s \in S$. This is quite intuitive, simply stating that the best policy is achieved by selecting the action that maximizes expected return in every state. As mentioned previously in this section, this is referred to as a greedy policy. Both eq. (2.45) and eq. (2.46) are deterministic policies, however as evident by eq. (2.33), policies are generally stochastic. Perhaps counter-intuitive, the *optimal* policy is not necessarily the *best* policy, meaning eq. (2.46) in particular needs some refinement. The reason for this, discussed in the next section, is the need for *exploration*.

2.5.4 Exploration vs Exploitation

Before discussing the concept of exploration, a qualitative comparison between eq. (2.45) and eq. (2.46) is warranted. Whereas eq. (2.45) relies on knowledge of both $R(s,a,s')$ and $T(s,a,s')$, eq. (2.46) relies only on the optimal action-value function. As mentioned in section 2.5.2, knowledge of the reward function and state transition function is not always available, as it requires some domain knowledge. The fact that a policy may be chosen without knowledge of the environment may seem surprising, but this is in line with the core idea of RL, namely learning by doing.

Algorithms aiming to calculate eq. (2.41) will have to do so by trying different actions, and recording the resulting rewards. However, as environments may be stochastic or rewards may even be delayed in time, having observed $Q_\pi(s_x,a_x) > Q_\pi(s_y,a_y)$ at a single instance, does not necessarily mean that this will always be the case. Keeping in mind that the optimal action is optimal with regards to what the *agent knows*, the value of knowledge should be considered. Generating more instances of unknown experiences e, as given in eq. (2.27), is thus necessary in order to correctly assess the value of an action. This introduces the need for exploring, that is trying actions outside the optimal policy. One way of implementing this, is by adjusting the policy eq. (2.46) so that it occasionally selects a random action, making it a stochastic policy as given by eq. (2.33). Selecting an action according to eq. (2.46) is known as *exploiting* or greedy behaviour, and should be used once the agent has learned its environment. After all, agents should ultimately maximize return, not explore the world.

This concludes the necessary background of classical RL. At this point, a discussion of RL would normally present different RL algorithms, ways of learning policies. In this case, the discussion moves to another branch of machine learning, traditionally viewed as separate from the field of RL, namely Deep Learning (DL).

2.6 Deep Learning

Deep Learning (DL) is a form of machine learning inspired by the mechanisms of the human brain. The material in this section is based on [28] and [29], unless otherwise stated.

2.6.1 Deep Feed Forward Networks

The quintessential DL model is the Deep Feed Forward Network (DFFN). Throughout this text, the term *network* will be used in reference to DFFN, which in turn is a class of Artificial Neural Network (ANN)s. The purpose of such a network is essentially to approximate an unknown function $\mathbf{y} = f^*(\mathbf{x})$, mapping some input $\mathbf{x}$ to some output $\mathbf{y}$. The network itself defines a mapping $\mathbf{y} = f(\mathbf{x};\theta)$.

The purpose of the deep learning process is to find the set of parameters θ that ensures that the network f approximates the function f^*.

The function defined by the network may be expressed as a n chained functions, i.e.

$$f(\mathbf{x}) = f^{(n)}\left(f^{(n-1)}\left(f^{(\ldots)}\left(f^{(1)}(\mathbf{x})\right)\right)\right). \tag{2.47}$$

Such a chain of functions is conveniently expressed as a *directed acyclic graph* of multiple layers. Each such layer corresponds to a function $f^{(i)}$ in the chain, taking as its input the preceding layers output, and feeding its output downstream. The number of such layers in a network is described as the *depth* of the network, hence the term deep learning and deep feed forward network. The output of eq. (2.47) is described as the *output layer*, the $\mathbf{x}$ the *input layer* and all layers in between are known as *hidden layers*.

It is convenient to define the output of a hidden layer as a vector quantity $\mathbf{h}^{(i)}$ corresponding to the i-th hidden layer, that is

$$f\left(\mathbf{h}^{(i-1)}\right) = \mathbf{h}^{(i)}. \tag{2.48}$$

Each hidden layer can be viewed as a collection of *nodes*, each node computing an entry to the layer output vector $\mathbf{h}^{(i)}$. Such a node constitutes the smallest computational units within a network, and consists of two steps:

1. The weighting of inputs and biasing
2. The activation of weighted and biased inputs

Keeping in mind that the input to a layer is a vector quantity, the input to each node within a layer may generally be the same list of numbers. In this case, the network is typically described as *fully connected* or *dense*. The general function implemented by a node, then, is given by

$$g\left(\mathbf{h}^{(i-1)\top}\mathbf{w} + \mathbf{b}\right) \tag{2.49}$$

where the function g is known as the *activation function*, and the quantities $\mathbf{w}$ and $\mathbf{b}$ are weights and biases respectively. The vector $\mathbf{h}^{i-1}$ is the output vector of the preceding layer, or equivalently, a vector consisting of the individual node outputs of the preceding layer. The parenthesis in eq. (2.49) is thus a weighted sum of the outputs of the preceding nodes, with an added bias. This weighted sum is then activated according to the activation function g. The activation function is essentially determining to what extent the particular neuron will feed its output forward in the network. This is somewhat analogous to how neurons in the brain are said to "fire" as a function of other neurons: If the weighted sum of inputs to the neuron is "strong enough", the neuron will register and fire to the next layer within the network. The terms "strong enough" and "fire" are formalized by the activation function.

Different activation functions exist, and choosing the appropriate activation function to use in the network depends on the particular application. A default

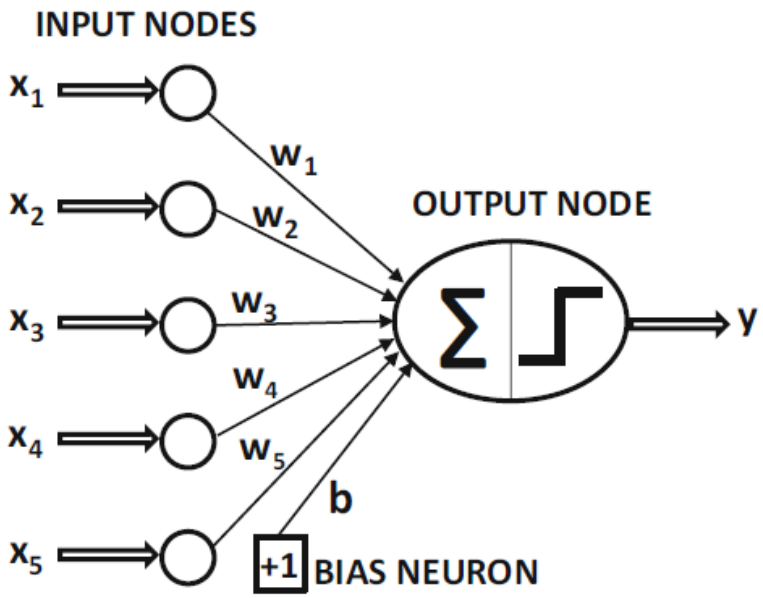

Figure 2.6: An illustration of a neuron. Figure courtesy of [28].

recommendation [30], which will also be utilized in this case, is the rectified linear unit (ReLu), given by

$$g(z) = \max(0, z) \tag{2.50}$$

in which case the neuron simply outputs its weighted sum of inputs, given that this sum is positive. The concept of a neuron is illustrated in fig. 2.6. Note the two parts of the computation of the output, the Σ and step-function indicating the summing of weighted inputs and activation respectively.

The network function eq. (2.47) is then fully defined by its *architecture* or *structure*, specifying the depth of the network, the width of each layer, the connections between nodes in consecutive layers and the activation function of each node, and its collection of weights and biases for each node. A simple network is illustrated in fig. 2.7

Recalling the start if this section, the network function was said to be parameterized by a collection θ. This collection is precisely the collections of weights an biases of the entire network.

2.6.2 Training Neural Networks

The DFFN provides the framework for enabling deep learning. But the description of the network so far has no learning capabilities, but rather a generalized function approximator. As stated earlier, the goal of the network function is indeed to approximate some unknown function, such as e.g. a *classifier*. A classifier typically maps a set of *features* to a *class* and will serve as an illustrative example for this section.

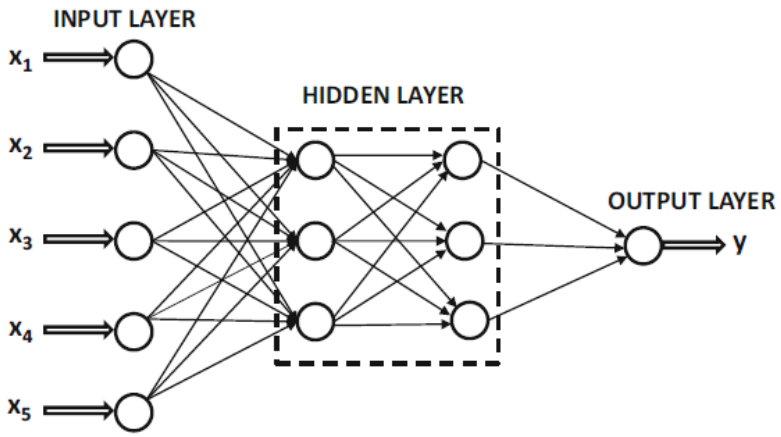

Figure 2.7: A simple neural network. Figure courtesy of [28].

It is clear that a deep neural network might implement such a classifier. Consider a function classifying a flower as either a rose, a tulip or a sunflower, implemented as a DFFN. The input to the classifier, i.e. the network, is a feature vector, where each entry represents some attribute of a flower. Such attributes could be width of the petal, length of the stem and area of the leaves. A feature vector describing a flower with a very long stem, narrow petals, and medium leaf area could be classified as a sunflower by setting the weights appropriately. The output $\mathbf{y}$ to such a network would typically be a vector with three entries, one for each class. The numerical value of the output would then correspond to the likelihood that the supplied feature vector was a description of the corresponding class.

In the case that the network does not classify correctly, but has the appropriate architecture, the classifier might predict that a description of a sunflower is a tulip. Noticing then, that the weight associated with e.g. the stem length is a low number for the sunflower output, one might adjust this number so that the network would have predicted the correct class for the same input.

This is essentially the way learning in an DFFN takes place, by comparing the prediction to the truth, and from this comparison adjusting the weights. What, then, is the point of an ANN, if the very thing it tries to predict has to be known in order for it to learn? This introduces the concept of *training*. The successful application of deep learning relies in sets of *labelled training data*, instances consisting of a feature vector with its correct label, also known as *target*. Such pair is denoted

$$\left(\mathbf{x}, \mathbf{y}^t\right) \tag{2.51}$$

where $\mathbf{y}^t$ is the label or target associated with the feature vector $\mathbf{x}$. The hope is that through this training, the labelled data has provided means for adjusting the weights to such a configuration that unlabelled data is predicted correctly.

The concept of training raises two questions: How is the prediction to be compared to the target, and given this comparison, how is it used to alter the

weights of the network?

The answer to the first question, how comparisons are to be made, is through a *loss function* on the form

$$L = L(\mathbf{y}^p, \mathbf{y}^t) \tag{2.52}$$

giving the *loss* of a prediction. As is the case with activation functions, the particular loss function to be used is an application dependent design choice. Intuitively, some distance function seems like a reasonable approach, like for instance the Mean Squared Error (MSE), which will be the only loss function discussed in this text. This is given by

$$L = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{y}_i^p - \mathbf{y}_i^t\right)^2 \tag{2.53}$$

where n is the number of labelled feature vectors. That is, rather than computing the loss of a single labelled feature vector, it is generally computed as an average over multiple such pairs. A set of labelled pairs for which the loss function is computed will be referred to as a *batch*, and the use of the network to predict a batch will be referred to as a *pass*. The quantity $\mathbf{y}^p - \mathbf{y}^t$ is known as the *error* $\mathbf{e}$ of the prediction. Loss is generally a function of error, i.e. $L = L(\mathbf{e})$

Once the loss is evaluated, how is this information used to adjust the weights in a reasonable way? Keeping in mind that the goal of this weight update is to change the network in such a way that the predicted output would have been closer to the target, the vector of weights should be changed in the direction that minimizes the loss function eq. (2.52) the fastest. This direction is readily obtained through the gradient of the loss function with respect to its weights. The procedure of minimizing the loss function this way is done using *gradient descent*. In order to compute the gradient of eq. (2.52), an algorithm known as *back propagation* is used. In the context of ANNs, this is known as the *backward phase* of the training. This implies the existence of a *forward phase*, which is simply the pass mentioned above. Backpropagation is further described in [31].

2.6.3 Convolutional Neural Networks

The Convolutional Neural Network (CNN) is a widely used class of neural network [28]. CNNs differ from the general neural network described above mainly in the number of connections between consecutive layers. Whereas the discussion of neural networks so far has assumed full connectivity, the hidden layer neuron of a CNN is only connected to a subset of neurons in the previous layer. This sparse connectivity lets CNNs learn features implicitly [32].

A CNN is a class of neural network for processing data with a grid-like topological structure. An example of such structured data is images, which can be viewed as a 2D grid of pixels. For this reason, CNNs have been very successful in computer-vision applications. As the name suggests, CNNs utilize a

mathematical operation known as a *convolution*. A CNN is by definition a neural network that uses a convolution in place of a general matrix multiplication in at least one of the layers [29].

2.7 Deep Reinforcement Learning

One of the goals of AI research is the development of fully autonomous agents, capable of improving performance over time through trial and error. As discussed in section 2.5, RL does indeed provide a principal mathematical framework for experience-driven autonomous learning. In practise, however, applying classical RL algorithms to develop highly sophisticated autonomous agents is infeasible, due to the fact that RL based approaches lacks scalability. These limits are the result of memory, computational and sample complexity, inherently limiting RL algorithms to small scale problems. This problem is often referred to as *the curse of dimensionality*, highlighting the vast computational and memory resources required to handle high-dimensional state spaces.

Deep Learning algorithms have provided the tools for overcoming these problems in RL. As discussed in section 2.6, the most important property of DL algorithms is their ability to find compact low-dimensional representations, or features, of high-dimensional data (e.g. text, images, audio). This addresses the curse of dimensionality associated with the practical application of RL. The application of DL algorithms within the field of RL defines the field of Deep Reinforcement Learning (DRL) [33].

2.8 Deep Q-Network

With the advent of DRL the possibility of controlling agents directly from high-dimensional sensory inputs became feasible. Previously, RL methods for accomplishing this task had relied on hand-crafting feature representations, i.e. states, from the high dimensional input for the specific problem. In addition to lacking generality, the performance of this approach is also limited by the quality of the feature representation.

In the the 2015 paper [17], the successful application of DRL-methods was demonstrated on a range of classical Atari 2600 video games implemented in the Arcade Learning Environment (ALE), allowing an agent to learn how play video games simply by observing the pixels of the game. In most games, the DRL-agent outperformed the best human players. The method presented and used in [17] is essentially using a deep neural network to approximate the the Bellman optimality action-value function eq. (2.43), described in section 2.5.2, repeated here for convenience

$$Q^*(s,a) = \sum_{s' \in S} T\left(s,a,s'\right)\Big(R\left(s,a,s'\right) + \gamma \max_{a'} Q_*\left(s',a'\right)\Big)$$

with a deep neural network. Once the approximation of eq. (2.43) is assumed to have converged, it can be used to implement an optimal policy given by using eq. (2.46), repeated here

$$\pi^*(s) = \text{argmax}_a \, Q^*(s,a).$$

This notion presents two particular problems. These problems, along with their solutions, are discussed in this section. Then the general algorithm for this approach, known as the Deep Q-Network (DQN) algorithm is presented. The algorithm takes its name from the deep neural network used to approximate the action-value function, which in the literature is often referred to as the Q-value function. The material in this section is entirely based on [34] and [17] unless otherwise specified.

2.8.1 Deep Q-Network Overview

The DQN computes the optimal action-value function iteratively, using the Bellman Optimality Equation as an iterative update rule. Noting that eq. (2.44) is expressed as a function of the environment transition dynamics, which generally are not available, it is perhaps more appropriate to reformulate it to

$$Q^*_{i+1}(s,a) = \mathbb{E}_{s' \sim S}\left[r + \gamma \max_{a'} Q^*_i\left(s', a'\right) \right] \tag{2.54}$$

where the added subscripts i is the i-th iteration. eq. (2.54) is also expressed in terms of expected value over the possible future states, instead of known rewards weighted by their probabilities, given by the model of the environment.

Furthermore, consider a parameterized approximation of eq. (2.54) given by

$$Q^*_i(s,a;\theta) \approx Q^*_i(s,a) \tag{2.55}$$

where θ is a vector of scalar parameters. Such an approximation can be made by a deep neural network, where the vector θ constitutes the weights of the network. As discussed in section 2.6, such a deep neural network can be trained to approximate the true action-value function $Q^*_i(s,a)$ by adjusting the weights over multiple passes.

Assuming the true value of eq. (2.54) was available for a given state and action s, a, this value would be a label y^t_i for the i-th update iteration $Q^*_{i+1}(s,a)$. The value produced by the approximation eq. (2.55) would then be considered the prediction y^p_i for this instance. Recalling the notion of the loss-function eq. (2.52), for instance by the MSE loss eq. (2.53) the loss is given by

$$L_i(\theta_i) = \left(y^t_i - Q^*(s,a;\theta_i)\right)^2 \tag{2.56}$$

for a single training instance. Note the slight change in notation between eq. (2.55) and eq. (2.56): the iteration index i is moved to the weights and loss function, as it parameterizes the optimal action-value function approximation at the i-th

iteration. Having obtained a loss, the weights are readily updated using back-propagation.

eq. (2.56) raises two important questions. Firstly, how are training instances $Q^*(s,a;\theta_i)$ to be generated? And secondly, how is the target value y_i^t to be obtained? Surely, if the true value of the optimal action value-function was known in the first place, the optimal policy would have been readily available, and one would not need to bother with training a deep neural network to reproduce the value. These questions are answered in the following section.

2.8.2 Experience Replay and Target Network

The answer to the first question posed at the end of section 2.8.1 is in line with the core idea of RL, namely that instances are generated through interacting with the environment. The Deep Q-network posed in section 2.8.1 is an approximation of the action-value function. The input layer to this network is a particular state $s \in S$, the output layer is a vector of action-values for that particular state, one node for each action $a \in A(s)$. As the same network structure is used for all states, it is assumed that $A(s) = A$. Training instances to the network, specifically instances of states, are thus generated through having the learning agent interact with the environment.

During training of a neural network, the training data should be fed to the network from a i.i.d. set. This is clearly not the case with states generated from interacting with an environment, as consecutive states depend on each other. Considering Atari Breakout, for instance, one would typically move the paddle the same direction for multiple consecutive states. If these states were to constitute the first batch of training, the network would favour moving the paddle left, for example, for all states, essentially constantly over-fitting for the last seen sequence of the game. This problem is addressed with a technique known as *experience replay*. Instead of training the network with instances as they appeared from interacting with the environment, these instances are stored as experiences, given by eq. (2.27), into a fixed length *replay memory* D_t of capacity $|D|$. The replay memory is subscripted with the time t, as new experiences are constantly added, and old experiences are pushed out. If the capacity of the replay memory is sufficiently large, a randomly sampled set B, known as a *minibatch* may be selected from the replay memory. This set of randomly recalled experiences constitutes the input to the Q-Network, and at each pass, a new set is sampled. The replay memory at time t is given by

$$D_t = \left\{e_t, e_{t-1}, \ldots, e_{t-|D|+1}\right\}. \tag{2.57}$$

With weights and states, the prediction of $Q^*(s,a;\theta_\mathbf{i})$ can be made. But how is the target value for this prediction calculated, or even understood? This problem arises as a result of combining supervised learning algorithms on unsupervised learning problems. The solution is essentially having the agent supply

its own label for a given training instance. This label is generated using a separate set of weights θ^-, being maintained in parallel with the neural network weights θ. This gives rise to a second neural network, with the same architecture as the Deep-Q network, but parameterized by a different set of weights. This network is used to calculate the label of each training instance and is aptly named the *target neural network*. The weights of the target neural network θ^- are copied from the Deep-Q network at regular intervals, every C episode.

2.8.3 The DQN Algorithm

With a description of the Deep-Q Network in place, the DQN algorithm can be described. The DQN algorithm builds upon the agent-environment framework, described in section 2.5.1. First, the agent observes the state of the environment and selects an action, either randomly, in order to encourage exploration, or by maximizing expected return, i.e. by eq. (2.46). The environment is affected by the action, returning a reward and a new state to the agent, in addition to information about whether or not the action terminated the episode. These quantities, that is $\{s, a, r, s', \text{done}\}$, constitutes an experience, which is stored in the replay memory. Next, given that the replay memory is sufficiently large, a mini-batch of experiences B are sampled from the replay memory. Then, the label

$$y^t = \left[r + \gamma \max_{a'} Q^* \left(s', a'; \theta_i^- \right) \right]$$

is calculated using the target neural network weights θ^-, for each experience in the mini-batch. Note that for experiences with the done-flag, that is experiences where the action terminated the episode, y^t is simply given by r (as there is no "next" value).

Next, all the states of the experiences in the mini-batch are fed to the Deep-Q Network, along with their labels y^t. The Deep-Q Network predicts a value for each action, given by its internal weights θ and the MSE loss is calculated for the entire mini-batch. Finally, the Deep-Q Network is ready to learn, by gradient descent for the weights.

This is done at *every single step*. Once the episode terminates, the environment is typically reset and the next episode starts. Every C step the target network weights are updated by $\theta^- \leftarrow \theta$. An overview of the training loop is given in line 1.

2.9 A* Path Findig Algorithm

The A* (pronounced "A Star") algorithm is a graph traversal and path finding algorithm. The algorithm, often considered as an extension of Dijksra's Algorithm, was first demonstrated in [35] and has been favored for its optimal efficient property [36], that is, no other optimal algorithm is guaranteed to

Algorithm 1: DQN Training Loop

```
Initialize environment;
Initialize Deep-Q Network weights θ;
Initialize target network weights θ⁻;
Initialize replay memory with set capacity D;
Require exploration chance ϵ;
for each episode do
    Reset environment;
    C = 0;
    while not done do
        s ← observe();
        σ ← random(0, 1);
        if σ > ϵ then
            a = argmax Q*(s, a)
        end
        else
            a = random-action()
        end
        (r, s′, done) = step(a);
        e ← {s, a, r, s′, done};
        Store e in replay-memory D;
        Sample mini-batch B from D;
        y ← [];
        x ← [];
        for e in B do
            Add s in e to x;
            Find target value of s in e using target neural network;
            Add this value to y;
        end
        Train Deep-Q Network using x as targets and y as labels;
    end
    if C ≥ reset weight interval then
        θ⁻ ← θ;
        C ← 0;
    end
end
```

expand fewer nodes than A*. This section briefly describes the workings of the A* algorithm, and how it is used to find an optimal path.

The purpose of the algorithm is to find a path between two *nodes*, a start node and a goal node, with the least cost. The cost may be considered as the length of the path, time spent, or any other relevant property to be minimized. For a mobile robot navigating a maze, for instance, the A* may be used to find the shortest path through the maze.

The algorithm starts by considering all nodes adjacent to its starting position. Using a grid world as an example, this is a set of four tiles, eight tiles if diagonal movement is allowed. The algorithm will then decide on which nodes seems most promising. This is done by calculating the cost of the path going through that specific node. This cost function is given by

$$f(n) = g(n) + h(n) \tag{2.58}$$

where $f(n)$ is the cost of node n, $g(n)$ is the cost going from the start node to node n and $h(n)$ is an estimate of the cost of going from n to the goal. For the grid world example, $g(n)$ would be the distance traveled from the start node to n, and $h(n)$ would be an estimate of the shortest path from n to the end node. It is important to note that $h(n)$ is indeed an estimate, an estimate which may be updated as the algorithm traverses more nodes.

From this set of adjacent nodes, the one with the lowest cost is selected. The other nodes are however still kept in memory, as they may have to be reconsidered. From the newly selected node, four (or eight) new adjacent nodes are made available, each with their own cost. These costs, along with the costs of the nodes that are kept in memory are compared, and another node is selected. So if none of the newly exposed nodes seem promising, the previously exposed nodes may be revisited and explored as a possible path. This procedure is repeated until the goal node is encountered.

More formally, at each iteration in the A* loop, adjacent nodes to the current node is considered and added to an *open list*. Then the node in the open list with the lowest score is selected, moved from the open list to a *closed list*. The selected node becomes the current node, and the previous node is defined as its *parent*. The reason for keeping open and closed lists and track of predecessor nodes is to maintain a memory of the unexplored "frontier" (the open list), nodes that have already been explored as a possible path (the closed list) and a link between which nodes lie on what paths (parenthood). So, once the goal node is found, the path can be determined by traversing back to the start node through each nodes parent.

Chapter 3

Methodology

As stated in section 1.2, the objective of the work presented in this thesis was to evaluate the application of DRL methods to learn and implement targeted sampling for an AUV in a simulated ocean environment. Specifically, by comparing the performance of DQN agent to that of the A* algorithm, and by discussing the challenges and potential advantages of this approach. To achieve this, an ocean environment was implemented as an OpenAI gym. A DQN agent as described in section 2.8 was implemented to interface with the environment. The agent was trained, resulting in a Deep Q-Network that would map states to actions. As a comparison, a model based approach was also developed using the A* algorithm. The performance of the DQN-agent and the A* algorithm was then compared on accumulated reward and encountered plankton.

3.1 Environment Model as an MDP

The simulation environment in which the agent acted is referred to as an ocean environment, consisting of a dynamic AUV simulation, described further in section 3.1.4, and a distributed biomass of plankton, described in section 3.1.5. In order for this environment to facilitate reinforcement learning, it was expressed as a finite MDP as described in section 2.5.2, that is, a collection of states and actions, known to the agent, as well as transition dynamics and reward function, unknown to the agent. This section describes how the ocean environment was modelled as an MDP.

3.1.1 States

The environment state consisted of the estimated ocean plankton density distribution, the AUVs relative position within the ocean, and the trajectory of AUV. In order to achieve this, the ocean was modelled as a square body of water, discretized into separate tiles. Each tile corresponded to a certain region within the ocean body. The area of the body of water was set to be $1000\,\mathrm{m}^2$

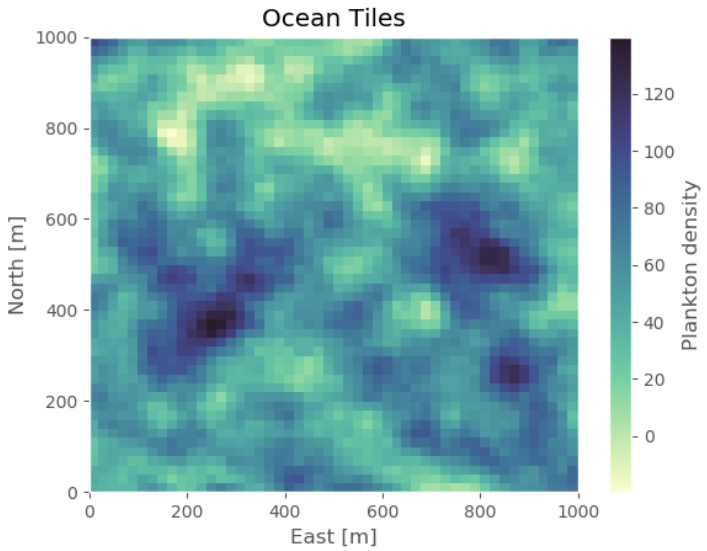

Figure 3.1: An example of the tiled ocean environment, displaying the plankton density of each tile.

and discretized into 50×50 tiles. A single tile τ, then, covered an area of $20\,\mathrm{m}^2$. A specific tile is referenced by its north-east coordinate, given in the NED-reference frame. For example, the tile $[37, 21]$, denoted $\tau_{37,21}$ refers to the square defined spanning from 740 m to 760 m north of the NED origin and from 420 m to 440 m east of the NED origin. The significance of this origin is arbitrary, but could for instance be considered the point from which the AUV was deployed.

Each tile $\tau_{x,y}$ is specified as a uple of three variables, the normalized plankton density ρ of that tile, a binary variable α denoting the presence of the AUV within that tile, and a binary value ν, denoting whether or not the tile was previously visited by the AUV. A single tile is then given by

$$\tau_{x,y} = [\rho, \alpha, \nu].$$

It was assumed that the plankton density was equal at all points within a tile. Furthermore, the set of tiles previously visited tiles $\tau_{x,y,}$ where $\nu = 1$ defined the trajectory of the AUV.

As an example, if the AUVs position is 741.2 m north and 437.3 m east of its deployment location, the α value $\tau_{37,21}$ is 1. The state space is the entire collection of tiles $\tau_{x,y}$ where $x, y \in [0 \ldots 49]$ for a total of 2500 distinct tiles. An example of such a set of tiles is shown in fig. 3.1.

3.1.2 Actions

Actions are the agents means of changing the state. Four actions were available to the agent, denoted N, S, E, W corresponding to the cardinal directions north, south, east and west. Keeping in mind that the DQN algorithm is model-free, the agent did not have any knowledge of what these actions in fact represented, or how they would change the state. The transition dynamics behind these actions were implemented as a form of waypoint generation. Each action would generate a waypoint corresponding to one of the adjacent tiles, specifically a waypoint 20 m to either direction. This waypoint was then received by the LOS-guidance system and motion control systems as described in section 3.1.4, initiating a guidance control loop. Once the waypoint was reached, the action was considered executed, at which point the agent would receive the resulting state, i.e. a new ocean map with the updated AUV-position, trajectory, and the plankton map. The plankton map was considered static, so the plankton map received at each step would not change. Additionally, the step would result in a reward, and information on whether or not the step terminated the episode.

3.1.3 Rewards and Termination

The reward function is a consequence of the agents interaction with the environment, and provides means for the ML-engineer to guide the agent to the desired behaviour. The desired behaviour was defined as following: Steer the AUV to the specific tile within the ocean map with the highest normalized plankton density. If small detours through patches with higher plankton is available, take these detours when seeking out the hotspot. The extent to which these detours should be made was not specified directly, but through different parameters in the algorithms and learning agents.

To implement this behaviour, the reward returned to the agent at each step was proportional to the change in distance between the AUV and hots-pot position. Thus, moving away from the hotspot would result in a negative change in distance, and yield a negative reward, also known as a penalty. Any action moving the agent towards the hotspot would close the distance, resulting in a positive change and consequently a positive reward. In order to encourage the agent to find the hotspot quickly, a constant penalty was imposed on each step. To achieve the aforementioned detour behaviour, rewards would also be given proportional to the encountered plankton density ρ of the tile containing the AUV after a step. To avoid having the agent steer the AUV back and forth between two adjacent high plankton tiles, and thereby exploiting the reward function, a reward for encountered plankton was only given if the tile was previously unvisited. Revisiting a tile would also give a penalty. This is also the reason the trajectory was included as part of state.

Finding the hotspot would immediately terminate the episode and yield a high positive reward. Additionally the mission would terminate if the AUV moved outside of the defined set of tiles, specifically over 1000 m to the east

Table 3.1: Values used for constants of the reward function of eq. (3.1).

Symbol	Definition	Value
P	Encountered plankton weight	2
D	Distance change weight	5
B	Revisit reward	-10
C	Move reward	-1
H	Hotspot reward	100
E	Exit map reward	-100
T	Waypoint timeout reward	-100

or north, or any distance south or west, from the map origin. Thirdly, the episode would terminate after a certain number of steps. Lastly, the episode would terminate if the AUV failed to reach the current active waypoint, specifically if the time spent reaching the waypoint exceeded a predefined waypoint timeout threshold. If the episodes terminated as a result of exiting the map or a waypoint timeout, a high penalty was returned to the agent. The reward function can thus be expressed as

$$R(s,s') = \begin{cases} R(s') = P\rho + D\delta + C & \text{if tile is unvisited} \\ R(s,s') = D\delta + C + B & \text{if revisiting tile} \\ H & \text{if tile is hotspot} \\ E & \text{if tile is out of bounds} \\ T & \text{if waypoint has timed out} \end{cases} \tag{3.1}$$

where ρ is the normalized plankton density of the tile containing the AUV, δ is the change in distance from the AUV to the hotspot. The values used for the constants and weights in eq. (3.1) are shown in table 3.1. Note that negative rewards are also referred to as rewards to avoid confusion in table 3.1, as the value is already specified to be negative.

3.1.4 AUV Simulator

The AUV dynamics was modelled using eq. (2.1). The AUV was simulated with a time-step of 0.1 using an ODE45 solver. The AUV simulator was implemented as a Python module with a step function as an interface to outside modules. The step function would accept a vector of control inputs commanding propeller thrust in Newtons, as well as elevator and rudder deflection in radians. The control input would then be saturated in accordance with the AUVs physical limitations, and low-pass filtered to avoid high frequent change in control surface deflection. The solver would then simulate the AUV dynamics for the next 0.1 seconds, update the states and make them available for other modules. The source code, including the parameters of the dynamics for the AUV

simulator was obtained from the AUV module of [1], with some minor modifications.

In addition to the AUV dynamics, LOS-guidance scheme and PID-control as described in section 2.3 and section 2.2 were implemented as additional Python modules. The source code for the controller was also obtained from [1], whereas the LOS-guidance scheme is original work. The guidance module would set waypoints in the order they were added to the waypoint database. When all waypoints were reached, the guidance state would be considered inactive, until new waypoints were added. The previous waypoint of the guidance scheme was at all times considered to be the AUVs current position, defining the reference path as the line connecting the AUV position and the next waypoint.

3.1.5 Plankton Data

The plankton data map was generated procedurally using a Gaussian Process, as described in section 2.4. This process would generate a $1000m^2$ map with a resolution of $20m^2$, resulting in a 50×50 grid of plankton distribution over an area, specifically the average density of plankton biomass in a given point in space. The kernel used in the Gaussian Process was the Matérn kernel, resulting in randomized smooth patches of higher plankton density, mimicking the distribution of biomass in the ocean. An example of a randomly generated plankton distribution using this process is shown in fig. 3.1.

Even though biomass in the ocean tends to drift with ocean currents, the plankton distribution was modelled as static for the duration of a single episode. Furthermore, the entire plankton map was assumed known and available as part of the state. Neither of these assumptions are realistic, and alternative approaches are discussed in chapter 5. The plankton hotspot was defined as the tile with the highest plankton density.

3.2 The Agent and the Algorithm

With the description of the environment given in section 3.1 rules of the game are set. This section describes the two solutions explored to achieve the desired behaviour. On the one hand, the ML approach utilizing the powerful capabilities of deep learning, on the other, the model based approach, using prior knowledge of the environment.

3.2.1 DQN Agent

The learning agent used to achieve this task was a DQN-agent. At the heart of this agent lies the DQN, which was implemented as a convolutional neural network. The design of the DQN agent is described in this section.

Table 3.2: Parameters used for agent replay-mamory.

Parameter	Value
Replay Memory Size	2048
Minimum Replay Memory Size	512
Mini-batch size	Entire Replay Memory
Discount factor	0.9
Target update counter	3

Recalling section 2.8.1, the DQN defines the policy of the agent, providing a mapping from an observed state to perceived values of each possible action in that given state. These values are calculated through a forward pass of the network. As such, the structure of the input layer corresponds to the state, in this case, a $50\times50\times3$ tensor. The input layer was followed by a 2D convolutional layer, with 32 filters and a 3×3 kernel. The layer was activated using the rectified linear unit function. After the activation, a max-pooling layer with a pool size of 2×2 was used. To reduce the effect of noise in the input data, a dropout layer was used after the pooling layer with a 10% dropout. These exact same three layers were then repeated, that is a convolutional layer, a max-pooling layer and a dropout layer. Before the output layer, which should correspond to the number of actions, the output from the last dropout layer was flattened, and then passed through a dense layer with 64 nodes. The output layer was then activated linearly, giving the estimated values for each of the possible four actions.

The parameters relating to the agents replay-memory, target update counter and discount factor is shown in table 3.2. These concepts are discussed in section 2.8.2 and section 2.5.2.

Note that instead of using a fixed-size mini-batch, the entire replay memory was fed to the network at each pass. This proved to give faster convergence, despite requiring more forward passes. The reason for this is not clear, but is discussed further in chapter 4.

3.2.2 A* Algorithm

The A* algorithm is not an agent. Unlike the DQN agent, it would evaluate the initial state of the environment once, and calculate the optimal path from the starting node, given by the AUV initial position to the goal node, given by the plankton hotspot using the A* algorithm as described in section 2.9. To achieve the detour behaviour described previously, a modification was made to the A* algorithm. The cost of the node, given by eq. (2.58) was given in terms of euclidean distances to encourage the algorithm to find the shortest path. However, a cost was added to each node as a function of the normalized plankton density of the node, specifically eq. (2.58) was given as

$$f(n) = g(n) + h(n) + P(1-\rho(n)) \tag{3.2}$$

where P is a positive constant, $\rho(n)$ is the normalized plankton density of node n. By this modification, tiles with low plankton densities would be given a higher cost, causing the algorithm to avoid them when faced with a tile with higher plankton density, even though it might cause a detour.

3.3 Implementation and Software Organization

As previously mentioned, the environment was implemented as a OpenAI gym and written in Python. The entire software is available at [21]. The environment package consists of three modules, the ocean environment, the plankton interface and the AUV interface. The software structure and interfacing is outlined here.

3.3.1 Ocean Environment

The ocean environment module provided the interface to external agents and algorithms as described in section 2.5.1 through a step method, taking as input an action, and returning a state, reward and a flag indicating whether the episode terminated or not. In addition to serving as an interface to external agents, the ocean interface module managed and synchronized information from two other modules to calculate rewards, check for termination criteria, change the state and render a figure of the environment. These modules are the AUV interface and plankton interface.

3.3.2 AUV-Environment Interface

The AUV interface serves as an interface between the OpenAI gym ocean environment and the dynamically modeled AUV. The module would process the action of the step function further by converting an action to an appropriate waypoint, pass this waypoint to the AUV guidance system, which would in turn determine the desired heading. The AUV interface would then feed the heading reference to the controller module, which would return the appropriate control input. This input was applied to the AUV system and simulated by the AUV dynamic model until the waypoint was reached. The AUV interface would then return the updated AUV position back to the ocean environment module.

3.3.3 Plankton Interface

The plankton interface module would generate plankton maps and locate the hotspot within this map. As the plankton was assumed to be static, no further processing was required at each step, but the module was developed with this possibility in mind.

3.3.4 DQN Agent

The DQN agent module implements the structures needed for the DQN agent as described in section 2.8. This included two structurally identical convolutional neural networks, one for the target and one for the model, a replay memory with methods for adding experiences, and sampling mini-batches of experiences, and a method for training the neural network, as described in section 2.8.1. Additionally, to give the module the workings of an agent, an action selection method was implemented

3.3.5 A* Algorithm

The A* algorithm was implemented as a Python module, taking as its input a plankton map, and producing a set of waypoints connecting the AUV starting position to the plankton hotspot. For the sake of consistency in the software, the A* algorithm would also interface with the ocean environment gym. Instead of calling the step function, which would require an action, a separate method was made taking as its input a collection of way-points, which in turn was handled by the AUV interface and the guidance and control systems of the AUV package.

3.3.6 The Training Loop

The ocean environment and an DQN agent was initialized and trained for 50 episodes. The DQN agent was initialized with an empty replay memory, a model neural network and a target neural network. At the start of each episode, the environment was reset, specifically by

- Randomizing the AUV position within the map
- Generating a new plankton environment using GPR
- Identify the plankton hotspot of the map
- Setting the status of the episode as not done

For each episode and as long as the episode was not done, the agent would then observe the state, and select an action. The action selection was done using the model neural network by presenting the observed state as an input to the network, and selecting the action that yielded the highest value from the output of the model neural network. The algorithm would select a completely random action with a probability of 0.3 in order to encourage exploration, as explained in section 2.5.4.

The agent would then execute the action, and receive a new state, a reward and information on whether the action terminated the episode. The reward was calculated, unless the episode was terminated, in which case the reward was given based on the reason the episode was terminated. Recalling section 2.5.2, this constitutes an experience on the form eq. (2.27). This experience was then added to the replay memory of the DQN agent.

After the replay memory was updated, the agent would train its neural network with a single pass of a mini-batch of states from the replay memory. This step was skipped until the replay-memory had reached a certain size, specifically 512 experiences. The details of the training are described in section 2.8.1. After each pass and weight update, the target neural network would copy its weights from the model neural network, given that a set amount of steps had been taken since he last update. The state received from the environment would then be considered the current state, and the loop would continue. The output of the training was a neural network which could represent the policy of the agent.

3.4 Performance Evaluation

To evaluate and compare the performance of the DQN agent and A* algorithm, both approaches were tested on the same plankton environment maps, using the same starting position for each map. The performance was evaluated on a series of metrics, which is further discussed in chapter 4.

Chapter 4

Results

This chapter presents the results of the DQN training, along with a comparison of the performance during training with that of the A* algorithm when faced with the same plankton scenarios. Additionally, a comparison between the pre-trained DQN-agent and the A* algorithm is briefly discussed, when faced with a different seed than during training. As a final experiment, the results of the DQN agent when training on the same plankton scenario for multiple episodes is compared to the A* solution to the same scenario.

4.1 Results of training

The model was trained according to the methods described in chapter 3. To reiterate some of the most important parameters, the exploration rate, ϵ was a static 0.3 for the duration of the training which was 50 episodes. As the agent would not actually train unless the replay-memory was sufficiently large, the agent did not in reality train until about twenty episodes in.

The metrics recorded during training were the accumulated reward per episode, the accumulated normalized encountered plankton per episode, the steps taken per episode, and the sum of distances between the agent and the hotspot at each step per episode. The latter metric serves as an indication on how close the agent was to the hotspot during an episode, when evaluated in relation to the number of steps of that episode.

Due to the nature of the reward function and how each episode was initiated, the potential reward would vary from episode to episode. For instance, maps that would generate with overall higher concentration of plankton would have a greater potential for reward. Episodes that saw a greater initial distance between hotspot and AUV starting position would also have a greater potential for reward. For this reason, some ratios between the metrics mentioned above will also be presented in this section. Most plots will be presented along with a low-pass filtered version of the same data, to reduce some of the noise present from episode to episode.

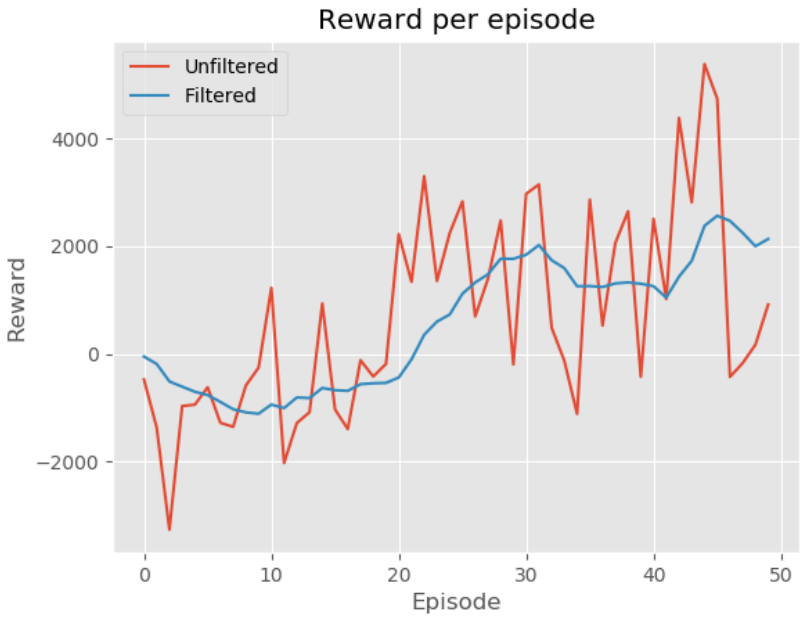

Figure 4.1: Accumulated reward per episode during training of the DQN-agent.

The overall reward accumulated at each episode during the training session is shown in fig. 4.1. The reward-to-step-ratio, that is, the accumulated reward per episode, divided by the number of steps of that episode, is shown in fig. 4.2.

These figures indicate a slight increase in performance over the course of the episodes. This increase is most distinct when comparing the first 17 episodes to the remaining episodes. Episode 17 marks the start of the agent using the replay memory to start training the neural network. This difference in performance is also clear from the reasons the episodes terminated during training. To reiterate, the possible termination criteria were finding the hotspot, leaving the map, reaching the step limit of 500 steps, or failing to reach the current waypoint. Before the neural network was used, all episodes terminated as a result of leaving the map. After using the neural network, the DQN-agent was able to locate the hotspot in 39% of the episodes, with the remaining episodes terminating by leaving the map. Of the episodes that failed to reach the hotspot, a substantial portion came within a few tiles of the hotspot. All the trajectories can be seen in appendix A.

Increase in performance between episode 18 and 50 is not that clear. Some high scoring episodes towards the end of the training gives the impression of a performance increase in fig. 4.1. When comparing to fig. 4.2, however, it is clear that the episodes in question were generated so that the potential for reward was high. Returning to the reward function eq. (3.1), the potential for reward is highest when the distance between the AUV and the hotspot is greater, as it

Figure 4.2: Accumulated reward relative to number of steps taken per episode during training of the DQN agent.

allows the most steps that closes the distance to the hotspot. The trajectories of the episodes in question, 44 and 45, are shown in fig. 4.3 and fig. 4.4. It should be pointed out that the hotspot was in fact not located in episode 45, as the AUV left the map right next to the hotspot.

To gain some insight as to how the reward function is aligned with the desired behavior, the encountered plankton, and encountered plankton-to-step ratio is shown in fig. 4.5 and fig. 4.6.

The encountered plankton data of fig. 4.5 indicate the same tendencies as the reward plots, that is, a clear increase after the agents has started training. Perhaps more interesting is the same tendency appearing in fig. 4.6, showing how much plankton was encountered on average each step of the episode. This plot indicates that maximizing the reward function eq. (3.1) also increases the plankton encountered per step. From these plots, episode 34 has a significantly high plankton count, both in terms of absolute and step relative values. Comparing to fig. 4.1, however, the reward is around 0. The trajectory for episode 34 is shown in fig. 4.7.

In episode 34, the AUV can be observed occupying high plankton tiles for most of the episode, without ever finding the hotspot. This example highlights a potential weakness of the reward function, namely that not enough reward is given for high plankton tiles. Returning once again to eq. (3.1) and table 3.1, the weight on approaching the hotspot is not only greater than the weight of encountered plankton, but typical values for a change in distance δ is much

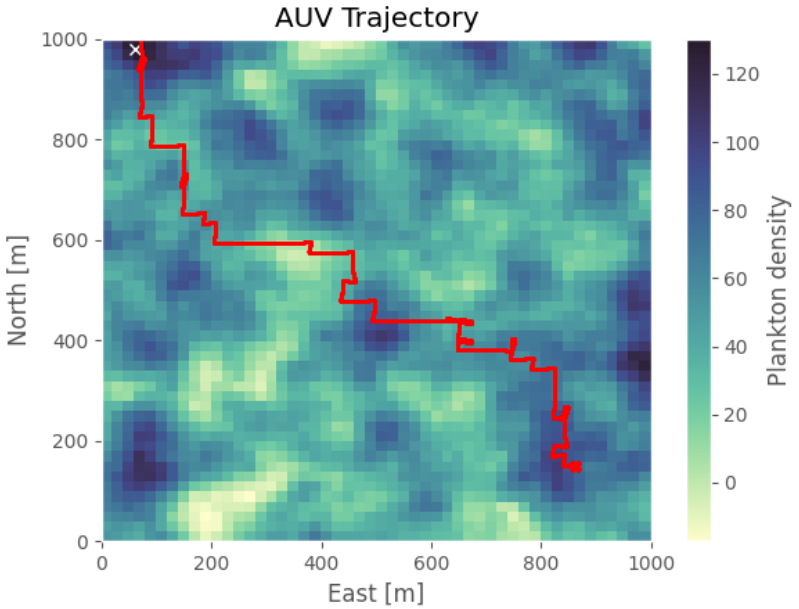

Figure 4.3: The trajectory produced by the DQN agent during training episode 44.

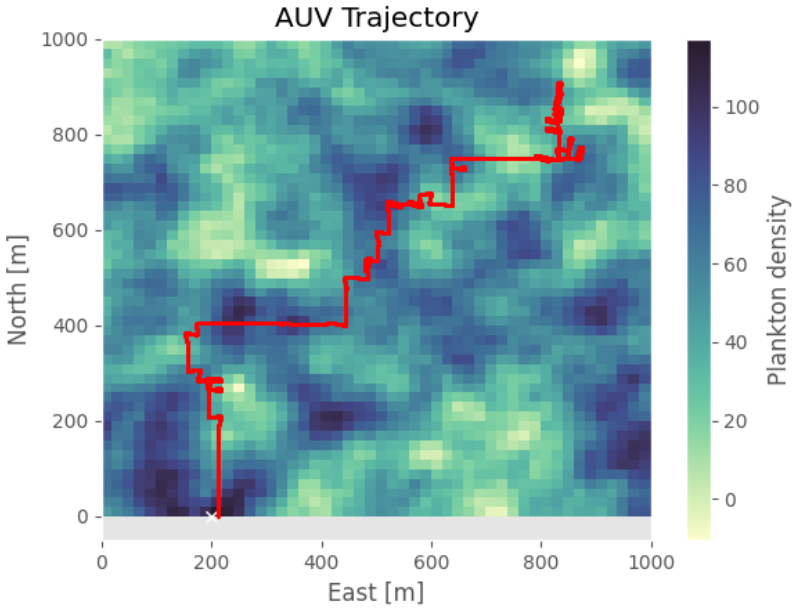

Figure 4.4: The trajectory produced by the DQN agent during training episode 45.

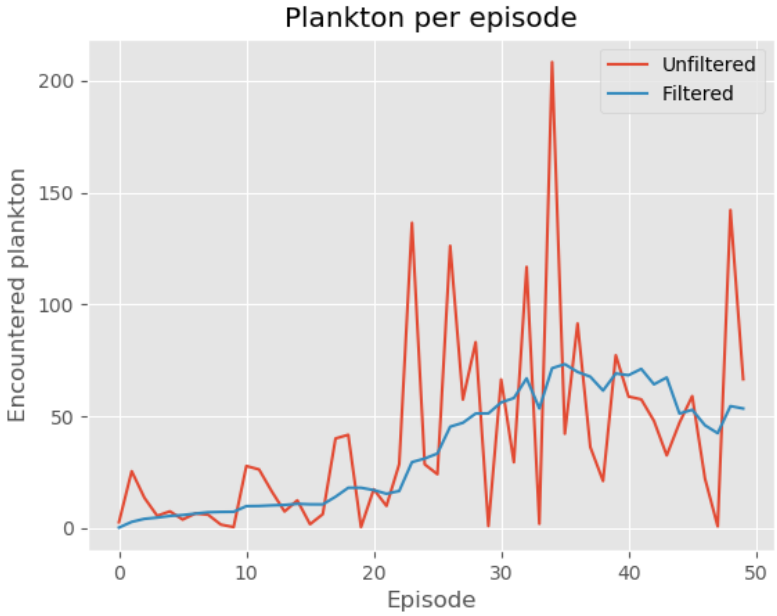

Figure 4.5: Accumulated encountered normalized plankton per episode during training of the DQN-agent.

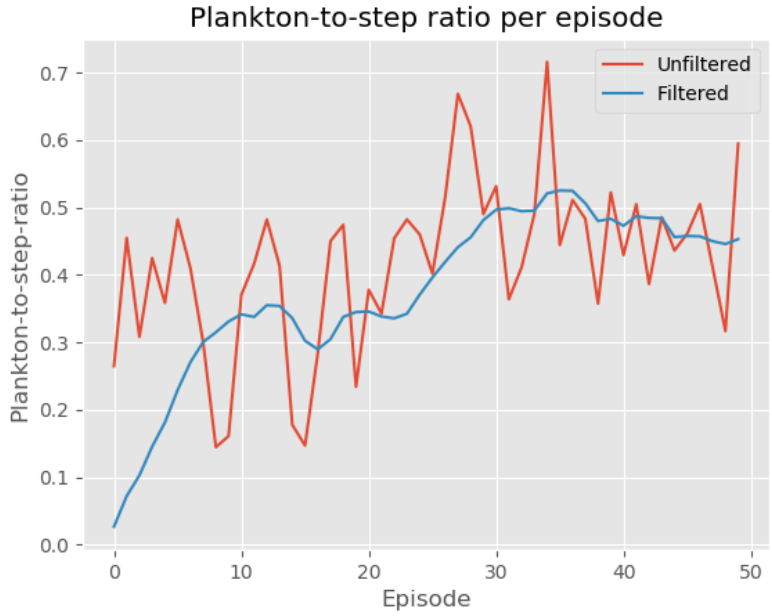

Figure 4.6: Accumulated encountered normalized plankton relative to number of steps taken per episode during training of the DQN-agent.

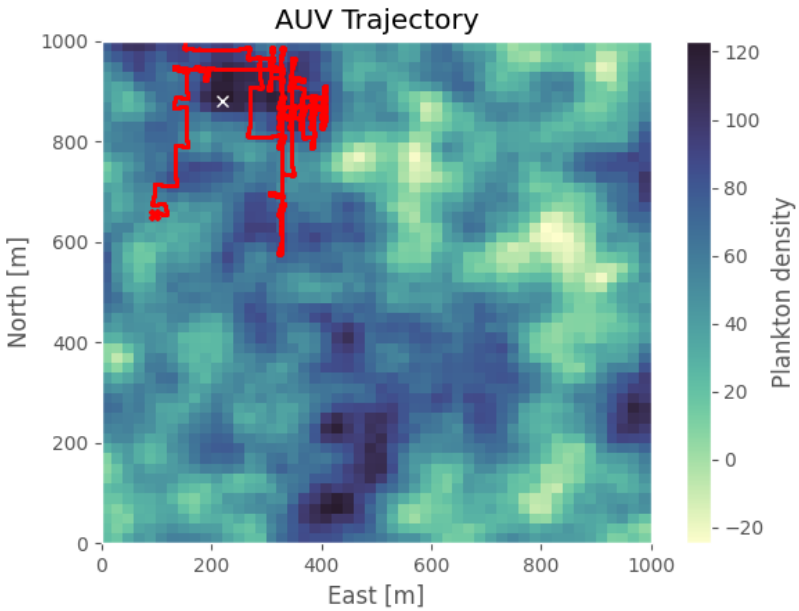

Figure 4.7: The trajectory produced by the DQN agent during training episode 34.

greater than a typical value for encountered normalized plankton ρ. A distance change lies in the range of $[-20, 20]$, corresponding to the size of a tile, whereas normalized encountered plankton δ lies in the range of $[0, 1]$

Finally, the ratio between accumulated AUV to hotspot distance and steps per episode is shown in fig. 4.8.

This plot shows the average distance between the AUV and the hotspot during an episode. So episodes where the agent steered the DQN close to the hotspot will have a lower value. Once again, an increase in performance is indicated before and after training. A notable anomaly is seen in episode 33. The trajectory is not included here, but can be found in appendix A. For this episode, the AUV position was initiated far away from the hotspot, and the AUV immediately exited the map after a few steps.

To reiterate the main point raised in this section, an increase in performance is clear before and after training. Whether or not the performance increased beyond this initial increase is not clear.

4.2 A* compared to DQN training

The 50 scenarios presented to the agent during training were also solved with the modified A* algorithm, as described in section 3.3.5. The same seed was used for the evaluation of the A* agent as for the DQN training session, mean-

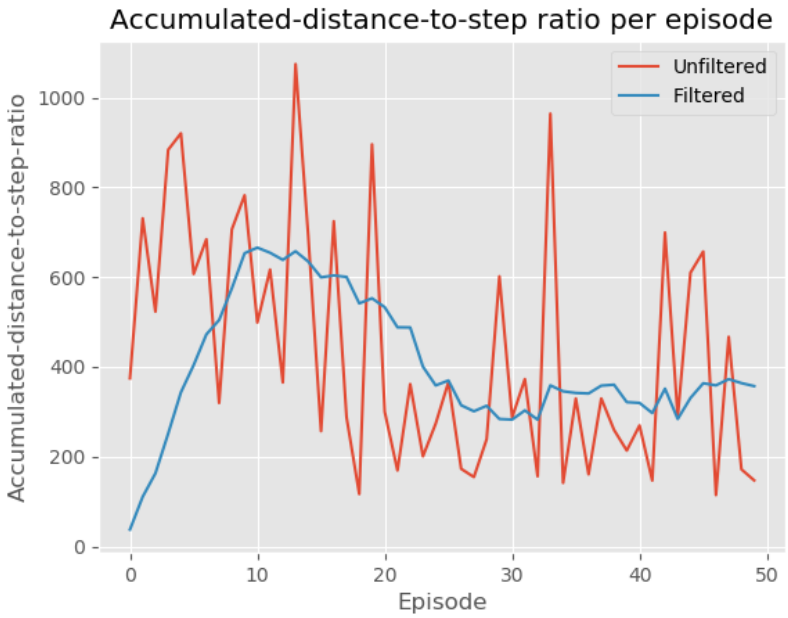

Figure 4.8: The average distance between the AUV and plankton hotspot per episode during training of the DQN-agent.

ing the algorithm and the agent were faced with the exact same plankton map and AUV starting positions. Rather than presenting the A* performance in separate figures, plots showing the ratio between the A* performance and DQN performance will be shown over the episodes. Thus, ratios above 1 indicate that the DQN outperformed the A* for that episode, whereas ratios below 1 indicates the opposite.

The main result of this comparison is the reward per episode from the resulting trajectories. It is important to keep in mind that the reward function is unknown to the A* algorithm. The A* algorithm was given rewards at each step according to eq. (3.1), that is, the same reward system used for the DQN algorithm during training. As discussed above, the number of steps taken per episode greatly affects the reward. For this reason, the reward ratio is shown relative to the steps taken for that episode. This result is shown in fig. 4.9.

This comparison reveals that the A* algorithm greatly outperforms the DQN agent. The slight increase in performance after training, however, is still evident in fig. 4.9. One anomaly in particular stands out. The trajectory of episode 16 is an example of the A* algorithm steering the AUV out of the map. Although the A* algorithm would never select a tile outside of the map, the AUV position may still end up outside any defined tile when the guidance system tries to reach the waypoint, thereby terminating the episode. This was the case for episode 16.

A comparison of encountered plankton per episode is shown in fig. 4.10.

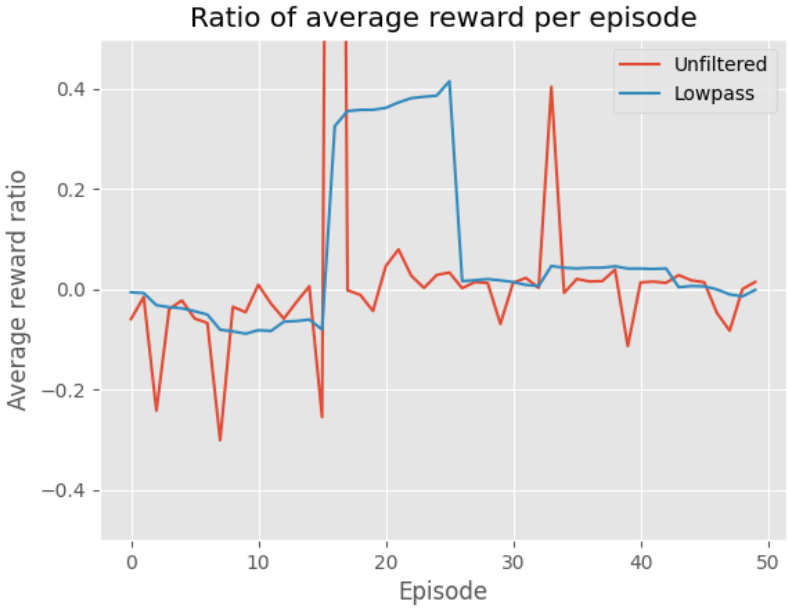

Figure 4.9: A comparison between the reward obtained per episode by the DQN-agent and A* algorithm during training. The comparison is shown as a ratio between DQN performance and A* performance, and is calculated relative to number of steps taken that episode.

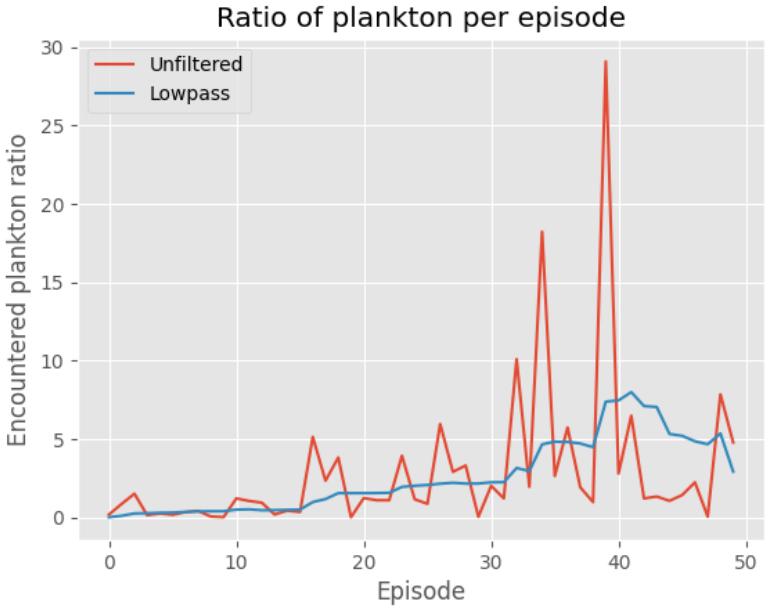

Figure 4.10: A comparison between the normalized plankton encountered per episode by the DQN-agent and A* algorithm during training. The comparison is shown as a ratio between DQN accumulated plankton and A* accumulated plankton.

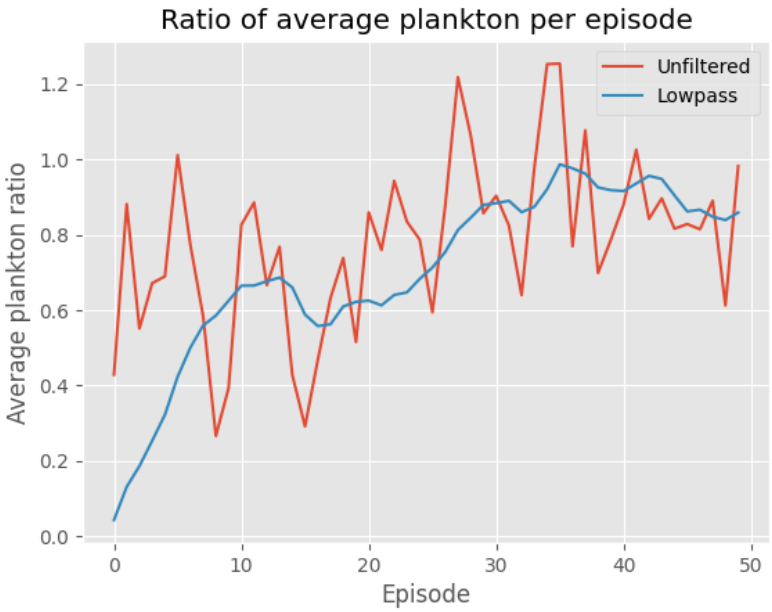

Figure 4.11: A comparison between the normalized plankton encountered per episode by the DQN-agent and A* algorithm during training. The comparison is shown as a ratio between DQN accumulated plankton and A* accumulated plankton.

From observing fig. 4.10 alone, the DQN-agent outperforms the A* agent with respect to encountered plankton. But, the DQN agent most often takes more steps than the A* agent, as the latter generally attempts to take the shortest path. To illustrate this, a comparison between average plankton per step is made in the form of a ratio between DQN and A*, shown in fig. 4.11.

From the encountered plankton point of view, the DQN algorithm approaches the A* performance during the course of the training. fig. 4.11 also illustrates the increase in performance after training with regards to plankton. Comparing this to the performance ratio with regards to reward fig. 4.9, it is once again clear that the reward function did not place enough value on encountering plankton, compared to reaching the hotspot. This point is discussed further in section 5.2.

To reiterate the main result of the comparison between A* and DQN performance, the A* algorithm greatly outperforms the DQN with respect to reward. A* also outperforms DQN when considering encountered plankton, but not by nearly as much.

This concludes the results from comparing the training of the DQN to the A* algorithm. Some trajectories of interest have been displayed here. All of the 50 trajectories can be found in appendix A for both A* and DQN.

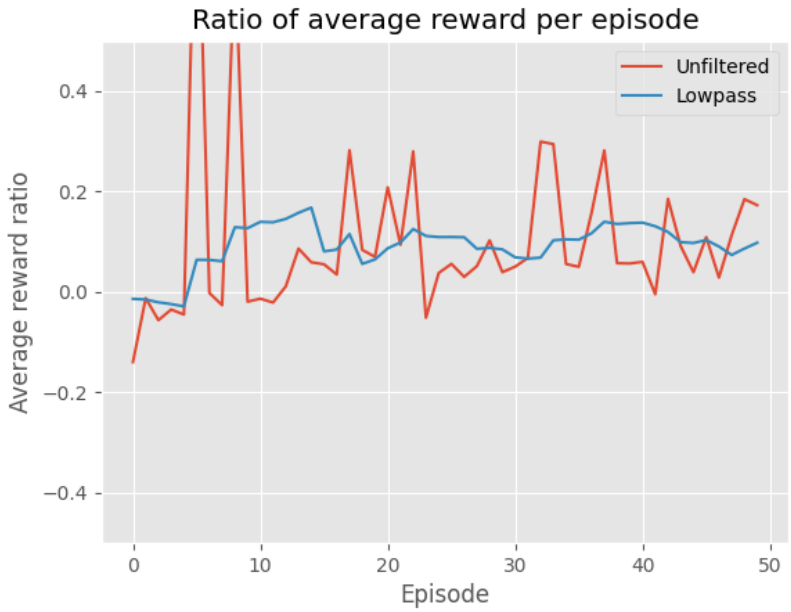

Figure 4.12: A comparison between the reward obtained per episode by the DQN-agent and A* algorithm during training on a single map. The comparison is shown as a ratio between the performance of the two, and is calculated relative to number of steps taken that episode.

4.3 DQN without resetting Map

As a final simulation experiment, the DQN agent was trained for 50 episodes on the same plankton map. The AUV starting positions were still randomized for each episode. For this experiment, the DQN agent was able to locate the hotspot in 95% of the episodes after the neural network was used.

Comparing with the A* algorithm, the DQN is still outperformed with regards to reward. A comparison of the average reward per step each episode is shown in fig. 4.12.

The performance ratio when the agent was able to train on a single map is marginally better than the ratio for multiple maps, seen in fig. 4.12, even if the agent was able to locate the hotspot more frequently. This is partly explained by the fact that the DQN agent takes more steps than the A* algorithm. This is in turn caused by the exploration chance ϵ present during training, which was set to 0.3 for the duration of the training. This exploration will occasionally result in missteps, moving the agent away from the hotspot, revisiting tiles, or leaving the map. This will not only increase the number of steps in many cases, but also reduce the overall reward.

Instinctively, a better test would be to compare the performance of the fully trained agent with the network obtained during training in a separate session,

and setting the exploration chance to zero. This did however result in a worse performance than the performance displayed during training. This was also the case for the pre-trained model having trained on 50 different maps.

This was due to the fact that the agent behaviour would frequently converge to alternating between two states at each step. The agent would steer the AUV back and forth between two adjacent tiles until the step limit was reached, and earning itself a massive negative episode reward in the progress. The reason for this fault is not clear, as the penalty for revisiting tiles was set quite high. One possible explanation might be that the agent did not get sufficient training in revisiting previous tiles, thereby not learning the connection between the state and reward. Keeping the random exploration chance would steer the agent out of this converging behaviour whenever it would occur during training.

A comparison between encountered plankton, not shown here, for the single map case showed similar results as that of fig. 4.10 and fig. 4.11.

4.4 Evaluating the Results

The results suggest that there is no evident benefit of the DQN agent developed in section 3.3.4 over the A* algorithm. Although the agent was able to locate the hotspot in some of the cases after having been trained, it would in most cases miss the hotspot and exit the map in the process. Comparing the performance of the trained model on the test set to that of the performance displayed during training revealed a converging behaviour where the agent would get stuck between two tiles Ideally, this behaviour would have been avoided, as it is associated with a high negative reward. The amount of revisits that occurred during training, however, was quite low, as evident from the AUV trajectories.

This points to insufficient training of the DQN model. This is further supported when comparing the performance of the agent when training on the same map for multiple episodes. In this case, the DQN agent was in fact able to learn path finding from scratch, through observing the plankton map over multiple episodes, and match the performance of A* in terms of locating the hotspot. This is however not a viable solution as it defeats the purpose of ML, namely applying knowledge to unseen data. This represents a general challenge with any ML approach, namely the need for large amounts of training data. For oceanographic sampling, this is particularly challenging, as the amount of available data is limited. For a ML approach to be applicable to a real world scenarios, the oceanographic data should be as close to real world plankton data as possible. Gathering a sufficient amount of real world plankton data to serve as input to a ML algorithm seems unlikely. For this reason, efforts should be made to develop a stochastic oceanographic model.

Furthermore, a comparison between the encountered plankton and obtained reward revealed that the encountered plankton had little significance on the reward function. As such, a successfully trained agent would possibly ignore high plankton tiles, as the reward for closing the distance is always much

greater. This highlights another challenge with RL in general: Given some predefined objective, how is it to be implemented as a meaningful reward function.

- Creating a realistic simulated environment plankton environment to provide relevant and sufficient input for the training of the DQN.
- Defining a reward function that both reflects the desired behaviour and ensures the ML algorithm converges.

Given that a realistic implementation of the ocean environment is implemented and an appropriate reward function is developed, what advantages might the ML approach present? This is not clear from the results presented in this thesis. However, some improvement in performance was achieved, and since DRL approaches have proven successful in uncertain and complex environments, its applications to targeted sampling should not be overlooked.

Chapter 5

Conclusion

5.1 Summary

The work presented in this thesis has explored the application of DRL on targeted oceanographic sampling. The purpose of the project was to gain insight to the potential of DRL methods, specifically DQN applied to a simulated ocean environment.

To achieve this, an ocean environment consisting of procedurally generated plankton data, and a dynamically modelled AUV was developed as an OpenAI gym interface, and a desired behaviour within this environment was defined as locating a plankton hotspot. A learning agent using a DQN algorithm was developed to train and perform in this environment. As a basis for comparison, a traditional path finding algorithm using the A* algorithm was also developed and evaluated.

The comparison of these algorithms revealed no immediate benefit of the ML approach for this simulated environment. A review of the performance during the training of the DQN algorithm did however reveal that the learning agent was indeed able to learn and to some extent generalize to unseen data. To answer the questions posed in section 1.2:

- A comparison between the DQN agent and A* algorithm revealed that the A* outperforms the DQN on all metrics
- The two main challenges revealed with applying DQN to targeted oceanographic sampling are
    - Ensuring a realistic plankton environment model
    - Designing a reward function that reflects the targeted behaviour
- No clear advantages with the DQN approach was revealed. However, DRL approaches should be investigated further.

5.2 Future Work

This final section describes the possible next steps in applying DQN to targeted sampling.

5.2.1 Plankton Model

The most crucial steps in ensuring a realistic simulator lies in the plankton model. The work presented in this thesis is based on several simplifications, as well as relying on synthetic plankton data. Two main improvements to the plankton model are suggested here, firstly, taking steps to ensure the data is more realistic, and secondly, ensuring that the way this data is perceived by the learning agent is more in line with what one might expect for a deployed AUV.

A more realistic ground truth plankton data should first of all account for the fact that plankton drifts with the ocean currents. Thus, the plankton map part of the state as described in section 3.1.1 would realistically change with each step. This a temporal dimension in the path-planning problem, and the A* algorithm presented in section 3.2.2 would not necessarily work. A ML approach, however, would arguably be better equipped to deal with this, as it introduces both uncertainty and complexity to the problem. Secondly, the plankton data should be collected from real world environments. Collecting the amount of data necessary for a ML agent to converge is no easy task, and for this reason methods for augmenting the data time-series of plankton data should be used to expand the training set.

Given a realistic ground truth plankton model, the state presented to the learning agent should be more in line with what is to be expected in a realistic scenario. This could be achieved by estimating the plankton map, from some sparse prior samples of the ocean environment. With the drifting plankton as described above, these samples should be accompanied by a time stamp, as well as a measurement of the ocean current. With these samples then, an estimated plankton map may be constructed using regression. In fact, GPR as described in section 2.4 is indeed a suitable regression model for this task, which indeed has been applied in AUV targeted sampling [12].

5.2.2 Ocean Environment

To accommodate the more realistic plankton model described above, the ocean simulator should include currents. Additional oceanographic phenomena could also be included, as temperature or salinity. This would depend on the availability of time-series of these phenomena synchronized with the plankton data, and assumes the deployed system will have sensors for measuring these phenomena.

5.2.3 Target Behaviour

The desired behaviour presented in this thesis was defined as locating the plankton hotspot, and encountering areas of high plankton density along this trajectory. To what extent this secondary goal should outweigh the hotspot seeking behaviour was never specified. Future work should specify clearer desired behaviours, with accompanying reward functions. Once a realistic model is in place, however, an ML based approach will arguably present a highly generalized solution, supporting different behaviours to be achieved within the same environment. This would allow a deployed AUV to be used for different missions, depending on the research goal of the experiment. Should the purpose of the deployment simply be to sample as much plankton as possible, a reward could be given based on every encounter. If the goal is to create a highly informative map of the environment, rewards could be given for minimizing uncertainty, given some plankton map estimate as described above. Missions could even be a combination of multiple behaviours, allowing the agent to be opportunistic and adapt its behaviour.

Bibliography

[1] S. T. Havenstrøm, *Path-following and collision avoidance environment for drl control*, https://github.com/simentha/gym-auv, 2020.

[2] N. Swift, *Easy a* (star) pathfinding*, https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2, 2017.

[3] *Deep q learning and deep q networks (dqn) intro and agent*, https://pythonprogramming.net/deep-q-learning-dqn-reinforcement-learning-python-tutorial/.

[4] *Openai gym*, Accessed: 2020-01-06. [Online]. Available: https://gym.openai.com/.

[5] R. Steward, *Introduction To Physical Oceanography*. Gainesville, FL: University Press of Florida, 2008.

[6] N. A. Cruz and A. C. Matos, 'Adaptive sampling of thermoclines with autonomous underwater vehicles,' in *OCEANS 2010 MTS/IEEE SEATTLE*, (Seattle, WA, USA), IEEE, Sep. 2010, pp. 1–6. DOI: 10.1109/OCEANS.2010.5663903.

[7] S. Petillo, A. Balasuriya and H. Schmidt, 'Autonomous adaptive environmental assessment and feature tracking via autonomous underwater vehicles,' in *OCEANS'10 IEEE SYDNEY*, (Sydney, NSW, Australia), May 2010, pp. 1–9. DOI: 10.1109/OCEANSSYD.2010.5603513.

[8] C. R. German, D. R. Yoerger, M. Jakuba, T. M. Shank, C. H. Langmuir and K.-i. Nakamura, 'Hydrothermal exploration with the autonomous benthic explorer,' *Deep Sea Research Part I: Oceanographic Research Papers*, vol. 55, no. 2, pp. 203–219, Feb. 2008. DOI: 10.1016/j.dsr.2007.11.004. [Online]. Available: https://doi.org/10.1016/j.dsr.2007.11.004.

[9] A. L. Kukulya, J. Bellingham, R. Stokey, S. Whelan, C. Reddy, R. Conmy and I. Walsh, 'Autonomous chemical plume detection and mapping demonstration results with a cots auv and sensor package,' in *OCEANS 2018 MTS/IEEE Charleston*, 2018, pp. 1–6. DOI: 10.1109/OCEANS.2018.8604524.

[10] Y. Zhang, R. McEwen, J. Ryan, J. Bellingham, H. Thomas, C. Thompson and E. Rienecker, 'A peak-capture algorithm used on an autonomous underwater vehicle in the 2010 gulf of mexico oil spill response scientific survey,' *J. Field Robotics*, vol. 28, pp. 484–496, 2011.

[11] Y. Zhang, J. P. Ryan, B. Kieft, B. W. Hobson, R. S. McEwen, M. A. Godin, J. B. Harvey, B. Barone, J. G. Bellingham, J. M. Birch and et al., 'Targeted sampling by autonomous underwater vehicles,' *Frontiers in Marine Science*, vol. 6, Aug. 2019. DOI: 10.3389/fmars.2019.00415. [Online]. Available: http://dx.doi.org/10.3389/fmars.2019.00415.

[12] T. O. Fossum, J. Eidsvik, I. Ellingsen, M. O. Alver, G. M. Fragoso, G. Johnsen, R. Mendes, M. Ludvigsen and K. Rajan, 'Information-driven robotic sampling in the coastal ocean,' *Journal of Field Robotics*, vol. 35, no. 7, pp. 1101–1121, Jul. 2018. DOI: 10.1002/rob.21805. [Online]. Available: http://dx.doi.org/10.1002/rob.21805.

[13] J. Birch, B. Barone, E. DeLong, G. Foreman, K. Gomes, B. Hobson, S. Jensen, D. Karl, B. Kieft, R. Marin, T. O'Reilly, D. Pargett, S. Poulos, C. Preston, H. Ramm, B. Roman, A. Romano, J. Ryan, C. Scholin, W. Ussler, S. Wilson, K. Yamahara and Y. Zhang, 'Autonomous targeted sampling of the deep chlorophyll maximum layer in a subtropical north pacific eddy,' in *OCEANS 2018 MTS/IEEE Charleston*, 2018, pp. 1–5. DOI: 10.1109/OCEANS.2018.8604898.

[14] N. Kohl and P. Stone, 'Policy gradient reinforcement learning for fast quadrupedal locomotion,' in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, vol. 3, 2004, 2619–2624 Vol.3. DOI: 10.1109/ROBOT.2004.1307456.

[15] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger and E. Liang, 'Autonomous inverted helicopter flight via reinforcement learning,' in *Springer Tracts in Advanced Robotics*. Springer Berlin Heidelberg, 2006, pp. 363–372. DOI: 10.1007/11552246_35. [Online]. Available: http://dx.doi.org/10.1007/11552246_35.

[16] Y. LeCun, Y. Bengio and G. Hinton, 'Deep learning,' *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. DOI: 10.1038/nature14539. [Online]. Available: http://dx.doi.org/10.1038/nature14539.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski and et al., 'Human-level control through deep reinforcement learning,' *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. DOI: 10.1038/nature14236. [Online]. Available: http://dx.doi.org/10.1038/nature14236.

[18] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot and et al., 'Mastering the game of go with deep neural networks and tree search,'

Nature, vol. 529, no. 7587, pp. 484–489, Jan. 2016. DOI: 10.1038/nature16961. [Online]. Available: http://dx.doi.org/10.1038/nature16961.

[19] S. Levine, C. Finn, T. Darrell and P. Abbeel, 'End-to-end training of deep visuomotor policies,' *Journal of Machine Learning Research*, vol. 17, no. 39, pp. 1–40, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-522.html.

[20] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz and D. Quillen, 'Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection,' *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018. DOI: 10.1177/0278364917710318. eprint: https://doi.org/10.1177/0278364917710318. [Online]. Available: https://doi.org/10.1177/0278364917710318.

[21] I. H. Kingman, *Targeted oceanographic sampling enabled by dqn*, https://github.com/ivanhkingman/auv-dqn-advanced/tree/master-thesis-ready, 2021.

[22] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*, 1st ed. The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom: John Wiley & Sons Ltd., 2011.

[23] J. G. Balchen, T. Andresen and B. A. Foss, *Reguleringsteknikk*. 2016, ISBN: 978-82-7842-202-1.

[24] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. The MIT Press, 2006, ISBN: 026218253X. [Online]. Available: www.GaussianProcess.org/gpml.

[25] S. Banerjee, B. P. Carlin, A. E. Gelfand and S. Banerjee, *Hierarchical Modeling and Analysis for Spatial Data*. Chapman and Hall/CRC, Dec. 2003. DOI: 10.1201/9780203487808. [Online]. Available: http://dx.doi.org/10.1201/9780203487808.

[26] R. Graham, F. Py, J. Das, D. Lucas, T. Maughan and K. Rajan, 'Exploring space-time tradeoffs in autonomous sampling for marine robotics,' in *Experimental Robotics: The 13th International Symposium on Experimental Robotics*, J. P. Desai, G. Dudek, O. Khatib and V. Kumar, Eds. Heidelberg: Springer International Publishing, 2013, pp. 819–839, ISBN: 978-3-319-00065-7. DOI: 10.1007/978-3-319-00065-7_55. [Online]. Available: https://doi.org/10.1007/978-3-319-00065-7_55.

[27] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book.html.

[28] C. C. Aggarwal, *Neural Networks and Deep Learning, A Textbook*, 1st ed. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer International Publishing, 2018. DOI: https://doi.org/10.1007/978-3-319-94463-0.

[29] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[30] V. Nair and G. E. Hinton, 'Rectified linear units improve restricted boltzmann machines,' in *Proceedings of the 27th International Conference on Machine Learning*, (Haifa, Israel), J. Fürnkranz and T. Joachims, Eds., Omnipress, 2010, pp. 807–814. [Online]. Available: https://icml.cc/Conferences/2010/papers/432.pdf.

[31] D. E. Rumelhart, G. E. Hinton and R. J. Williams, 'Learning representations by back-propagating errors,' *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. DOI: 10.1038/323533a0. [Online]. Available: http://dx.doi.org/10.1038/323533a0.

[32] N. Aloysius and M. Geetha, 'A review on deep convolutional neural networks,' in *2017 International Conference on Communication and Signal Processing (ICCSP)*, (Chennai, India), IEEE, Apr. 2017, pp. 0588–0592. DOI: 10.1109/iccsp.2017.8286426. [Online]. Available: http://dx.doi.org/10.1109/ICCSP.2017.8286426.

[33] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, 'Deep reinforcement learning: A brief survey,' *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017. DOI: 10.1109/MSP.2017.2743240.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, 'Playing atari with deep reinforcement learning,' 2013. arXiv: 1312.5602 [cs.LG].

[35] P. E. Hart, N. J. Nilsson and B. Raphael, 'A formal basis for the heuristic determination of minimum cost paths,' *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

[36] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, New Jersey 07458: Pearson Education Inc., 2009.

Appendix A

Appendix A: Every trajectory

This appendix lists all trajectories made by the A* algorithm and DQN agent from the comparison presented in section 4.1. Each episode is shown in a separate figure, with the performance the DQN agent on the left, A* on the right.

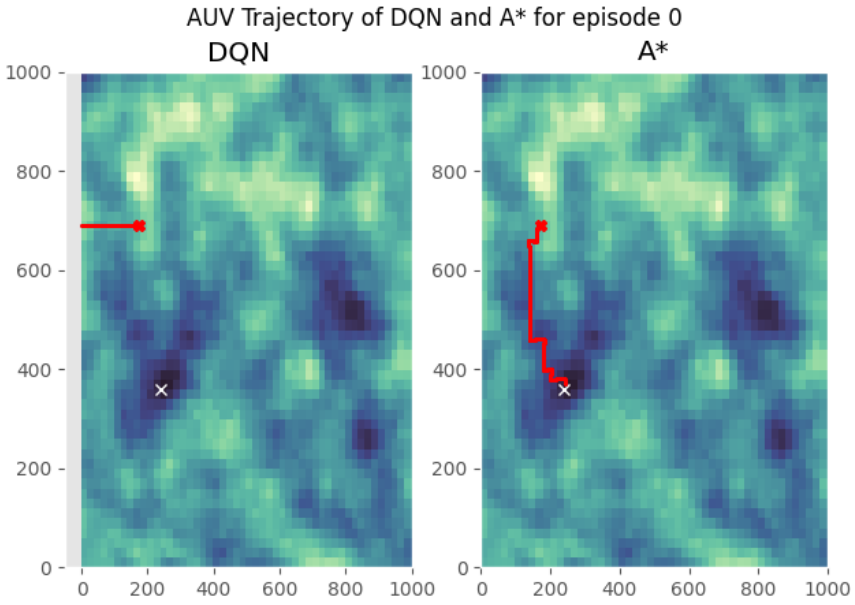

Figure A.1: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 0.

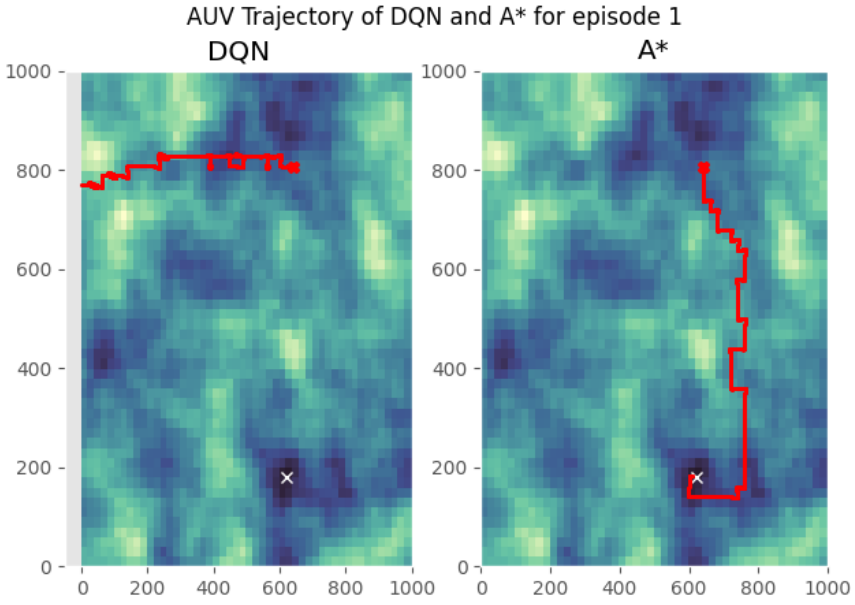

Figure A.2: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 1.

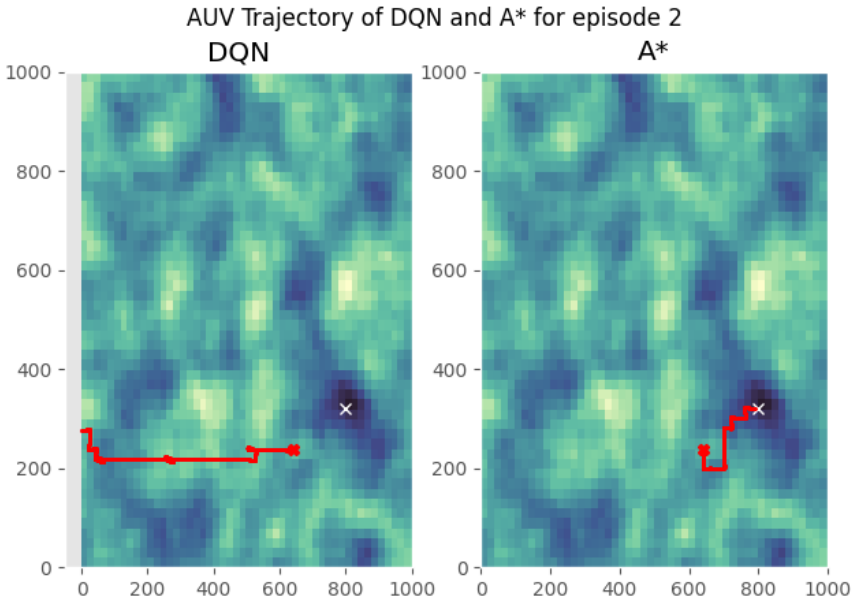

Figure A.3: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 2.

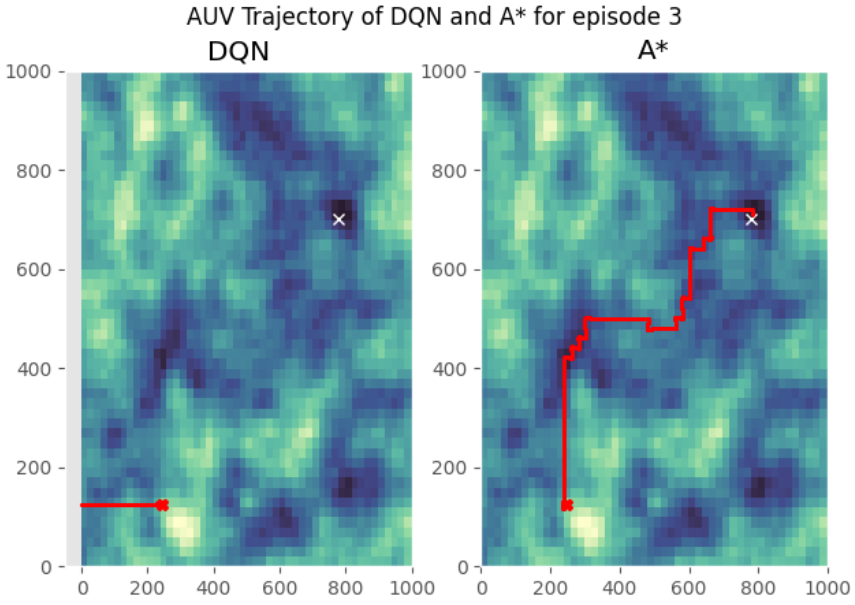

Figure A.4: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 3.

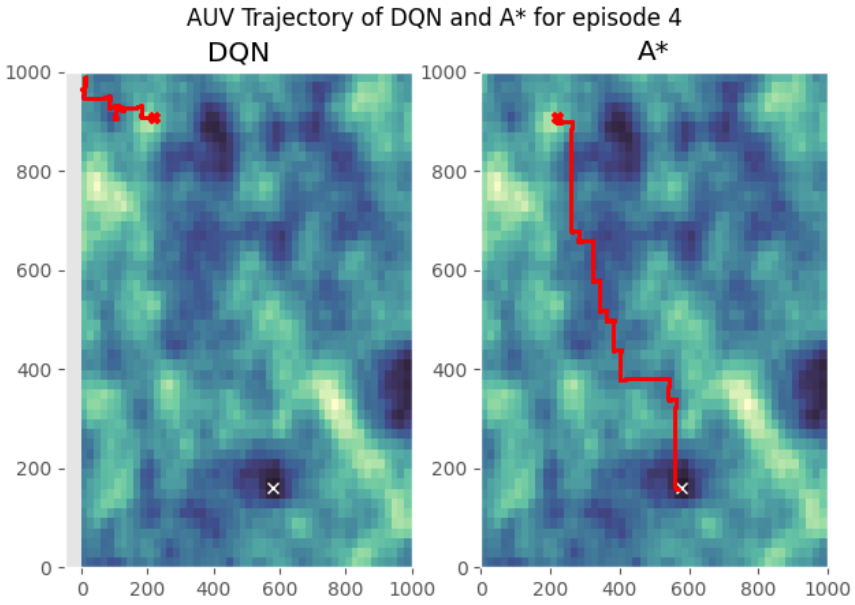

Figure A.5: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 4.

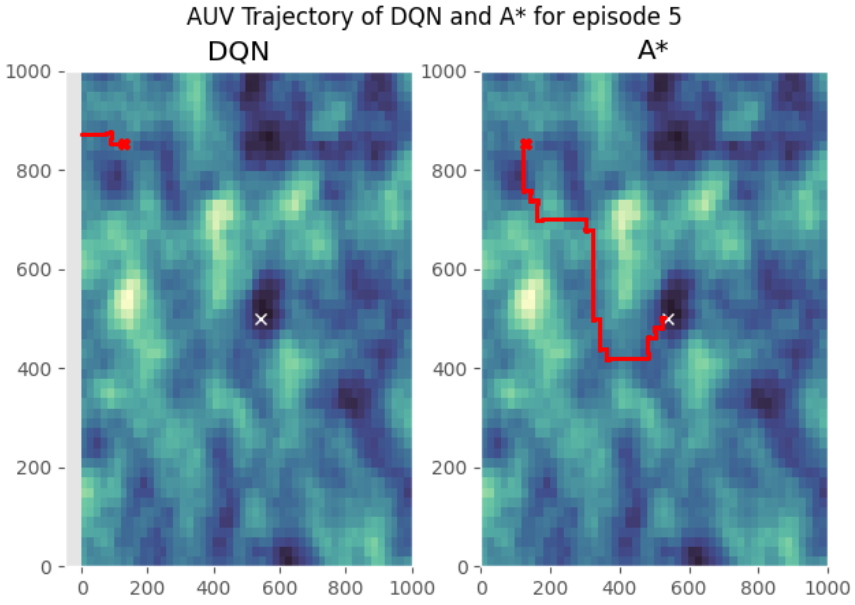

Figure A.6: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 5.

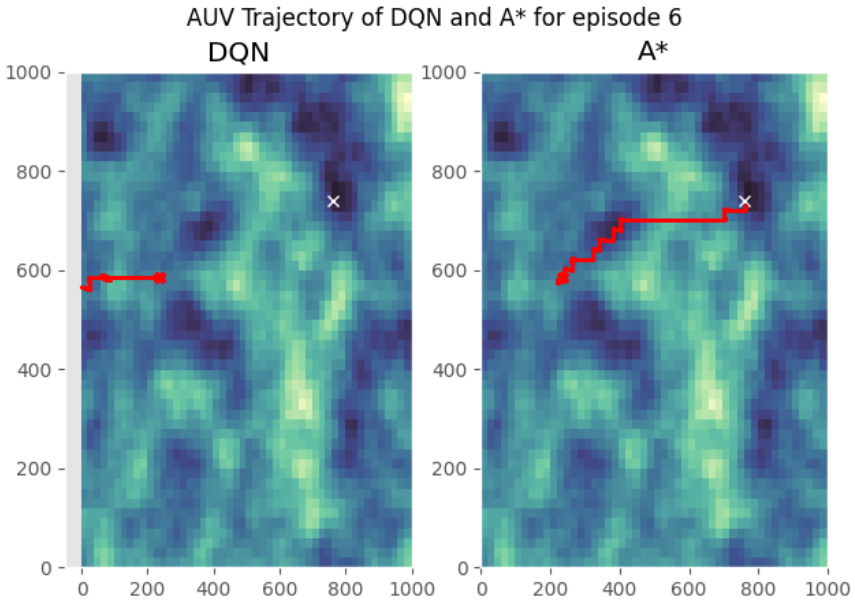

Figure A.7: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 6.

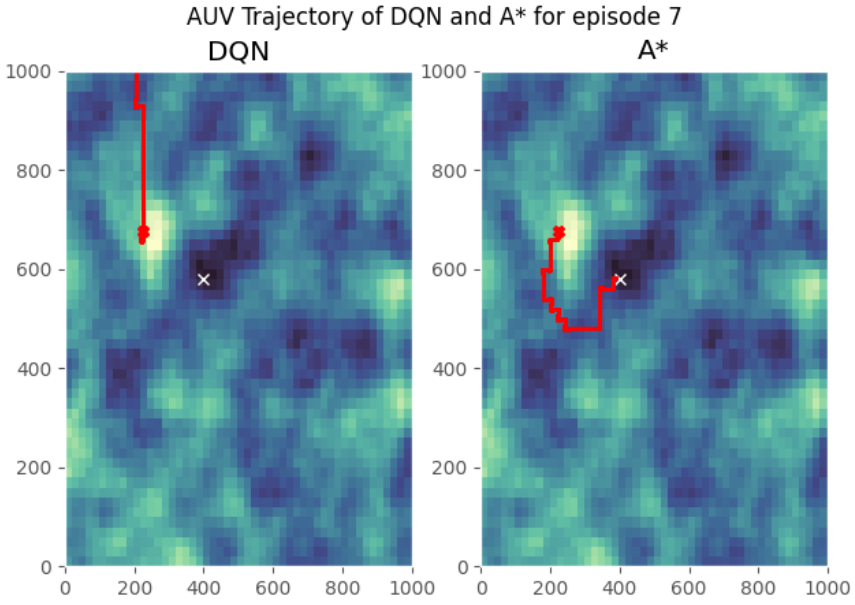

Figure A.8: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 7.

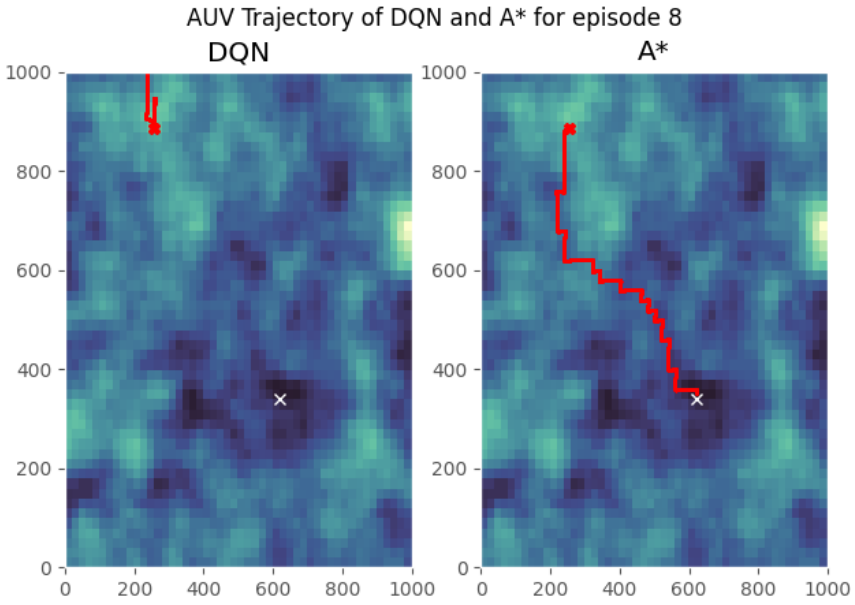

Figure A.9: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 8.

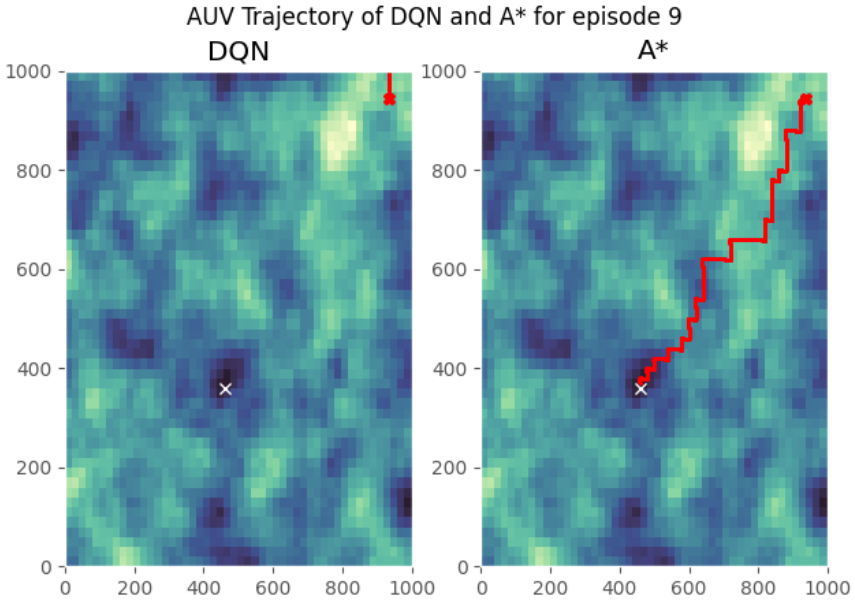

Figure A.10: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 9.

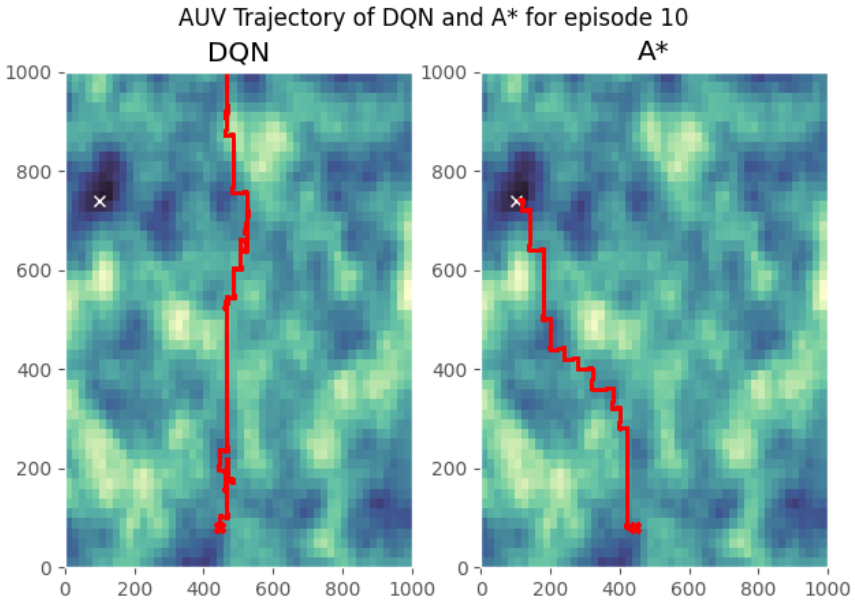

Figure A.11: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 10.

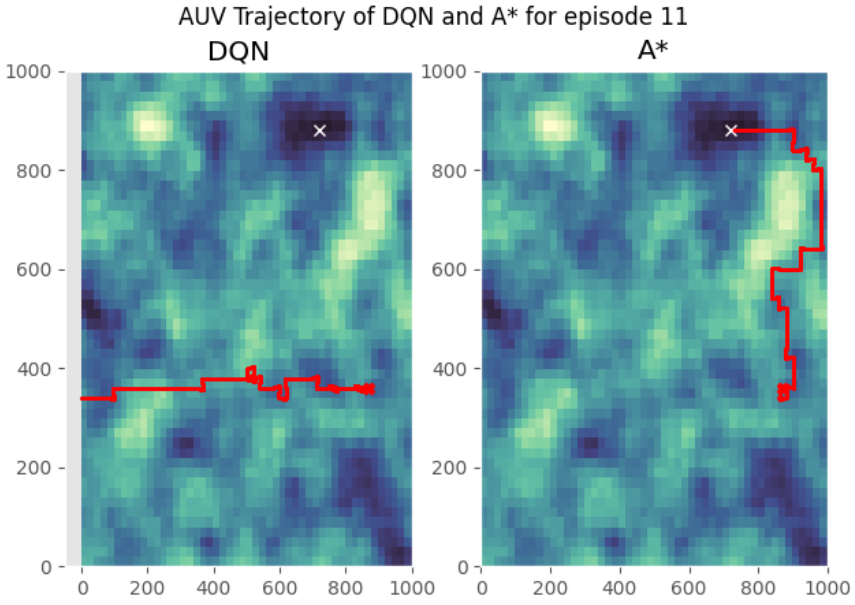

Figure A.12: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 11.

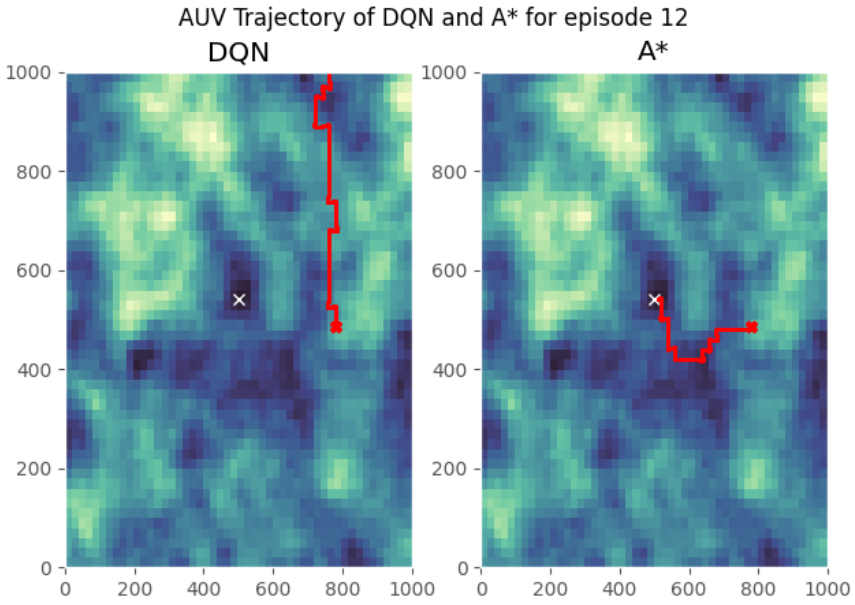

Figure A.13: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 12.

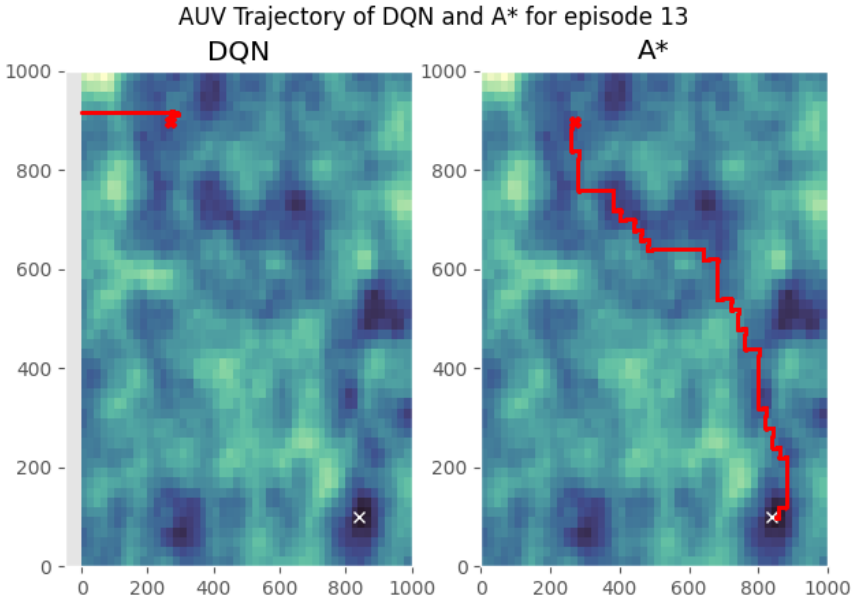

Figure A.14: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 13.

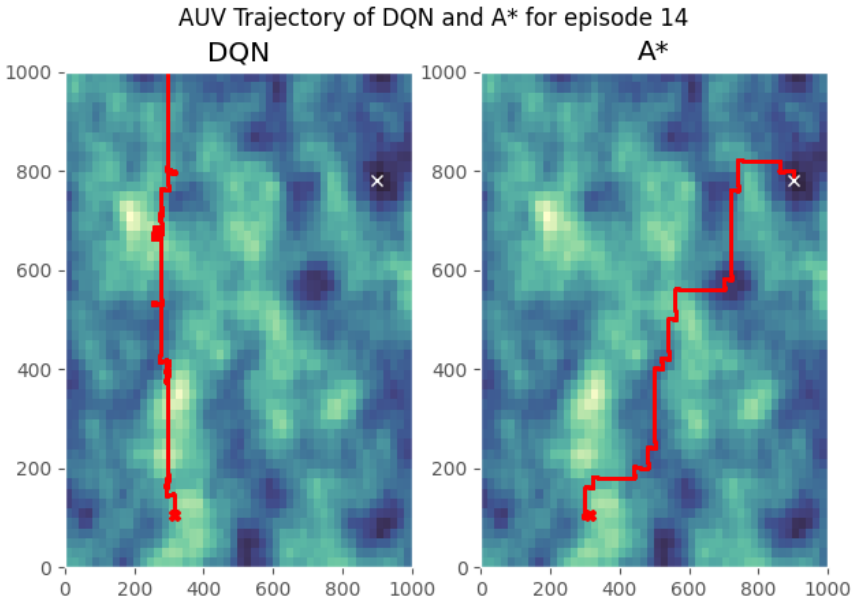

Figure A.15: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 14.

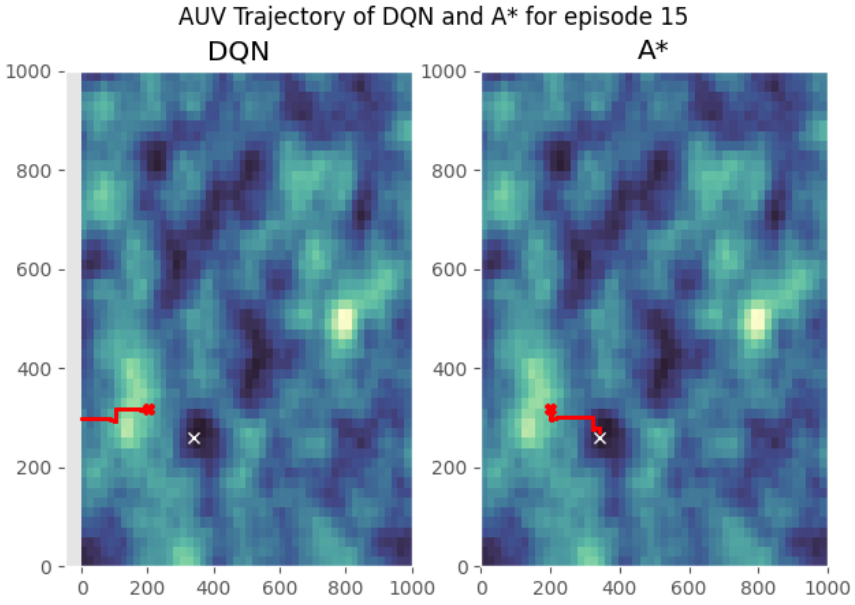

Figure A.16: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 15.

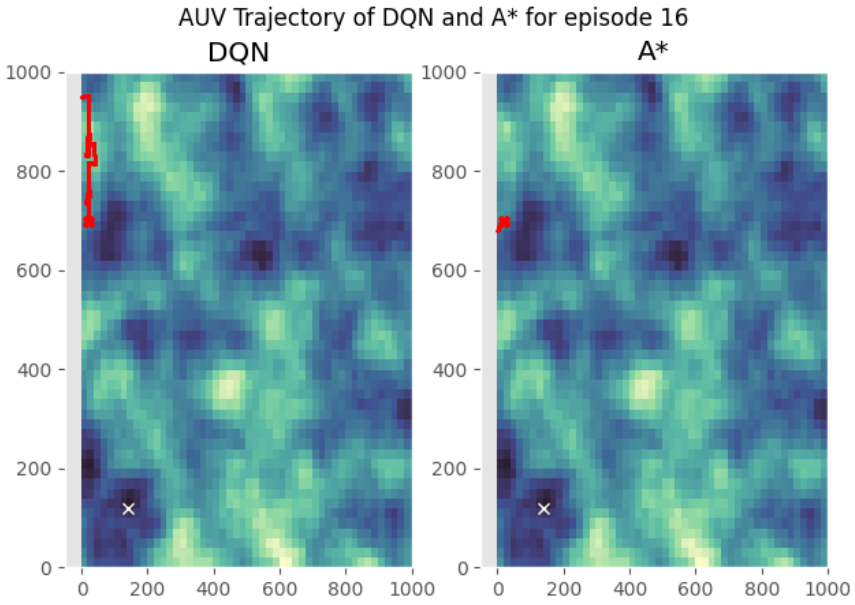

Figure A.17: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 16.

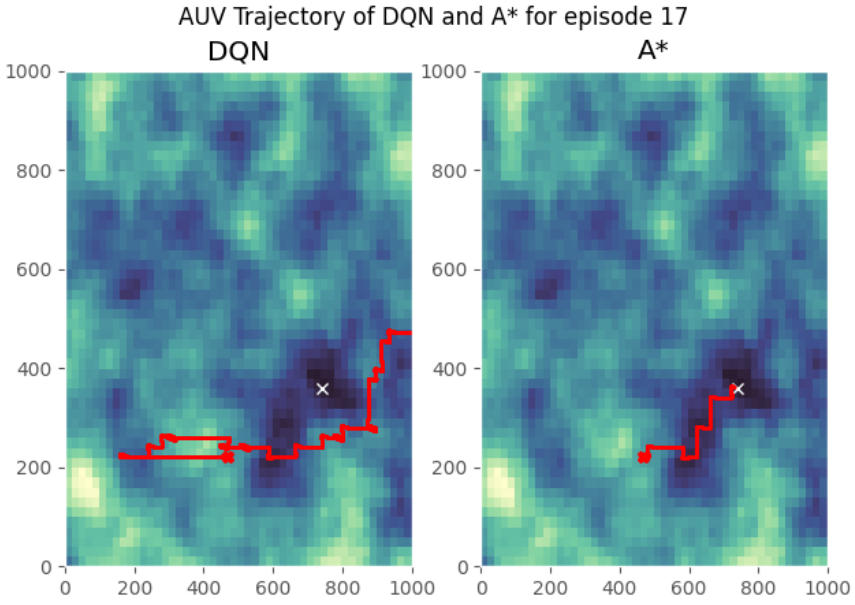

Figure A.18: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 17.

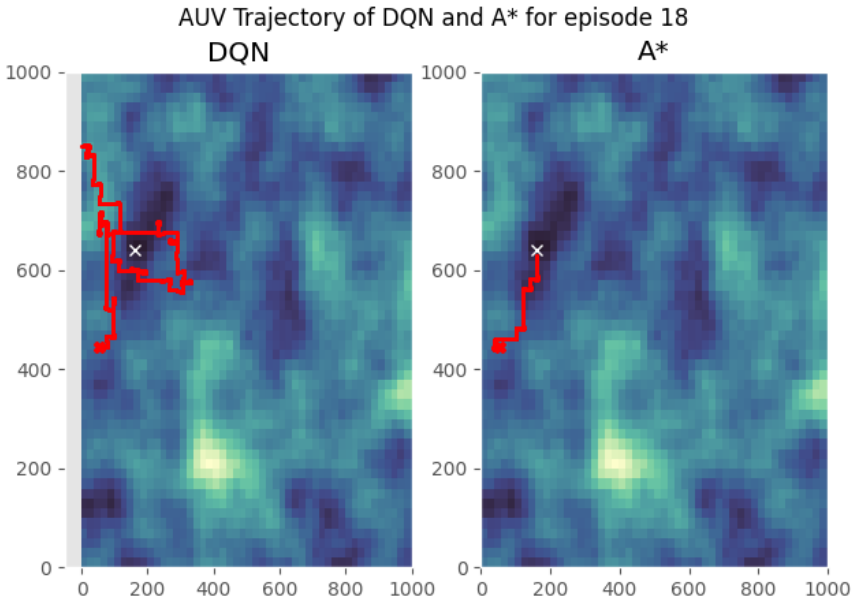

Figure A.19: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 18.

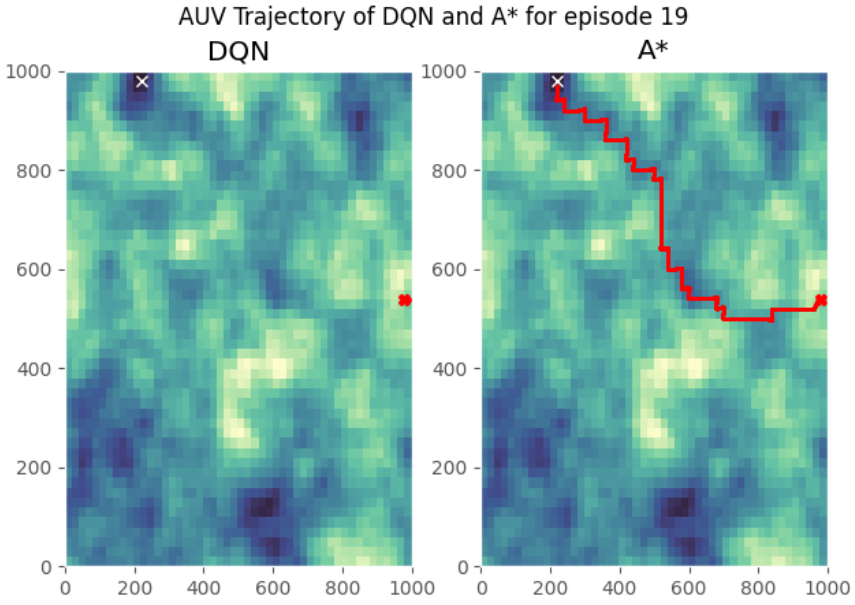

Figure A.20: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 19.

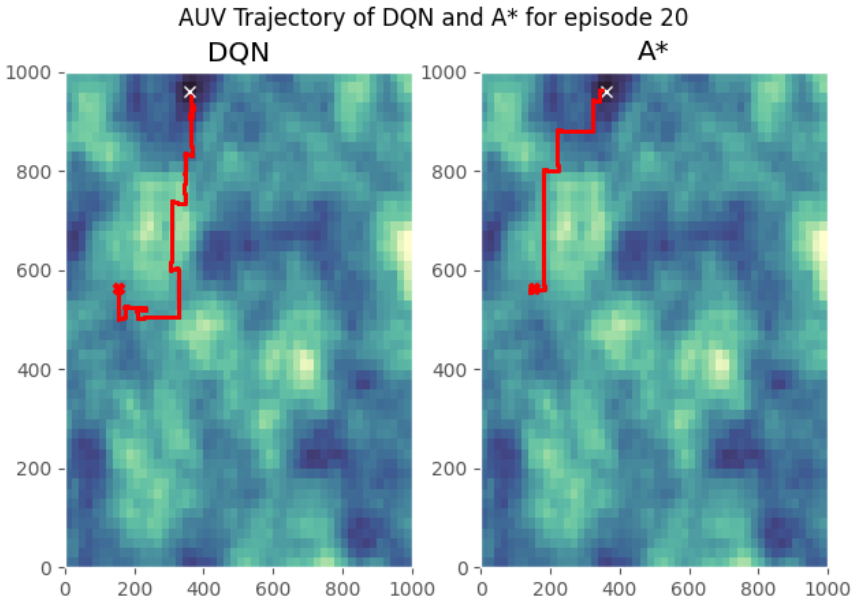

Figure A.21: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 20.

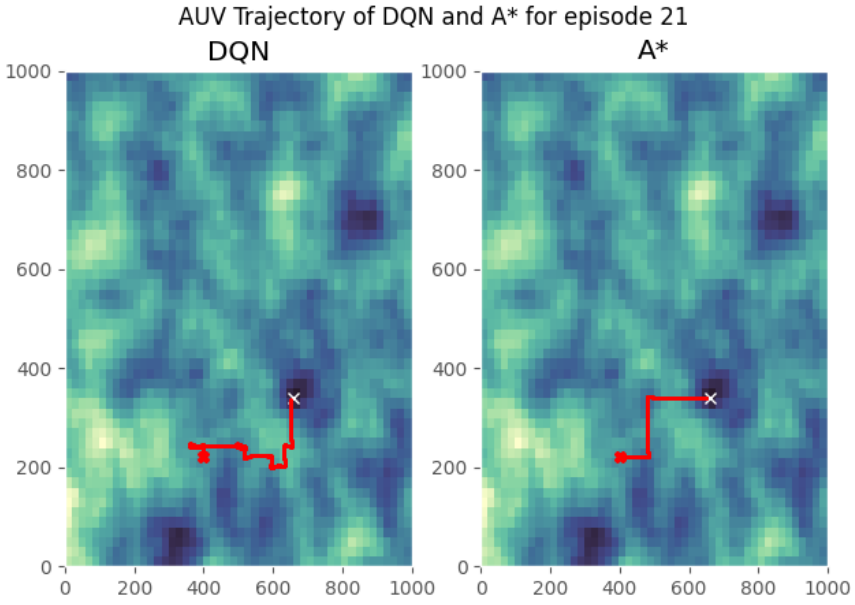

Figure A.22: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 21.

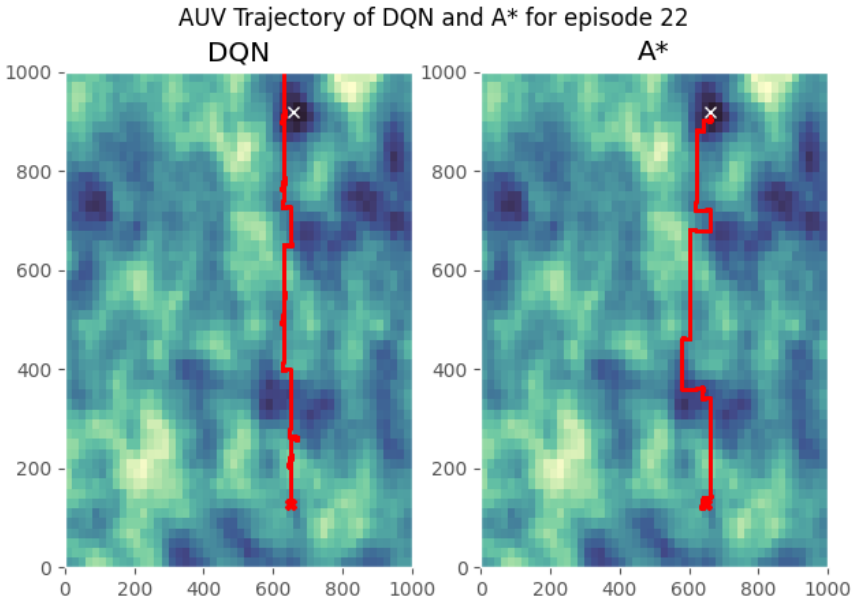

Figure A.23: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 22.

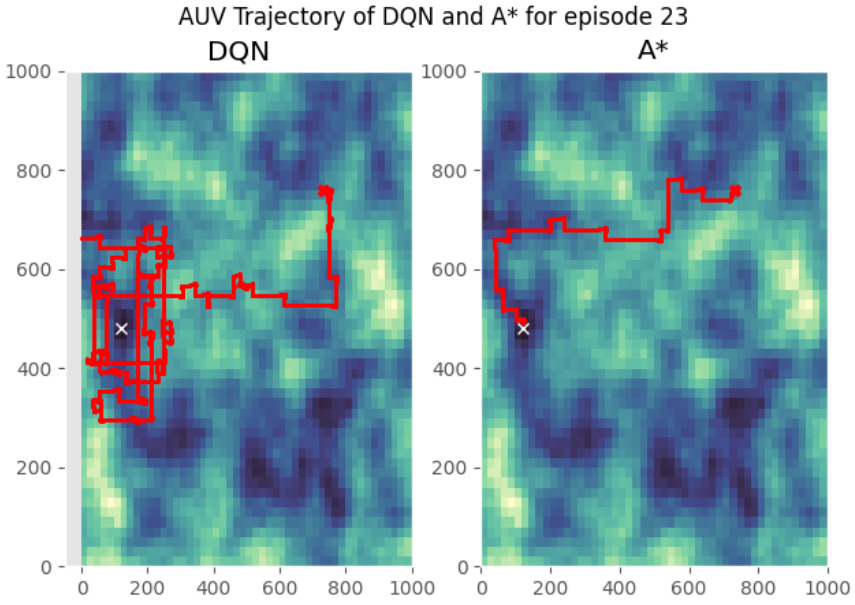

Figure A.24: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 23.

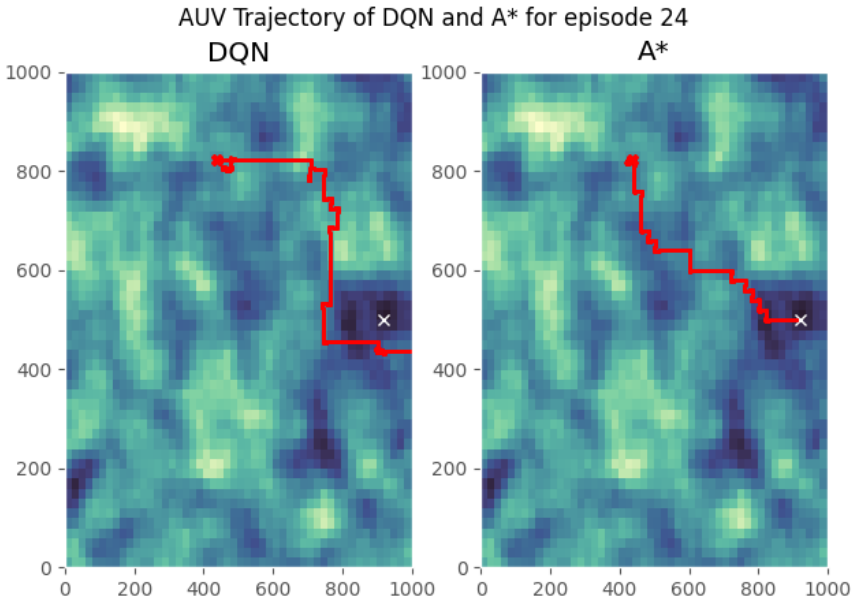

Figure A.25: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 24.

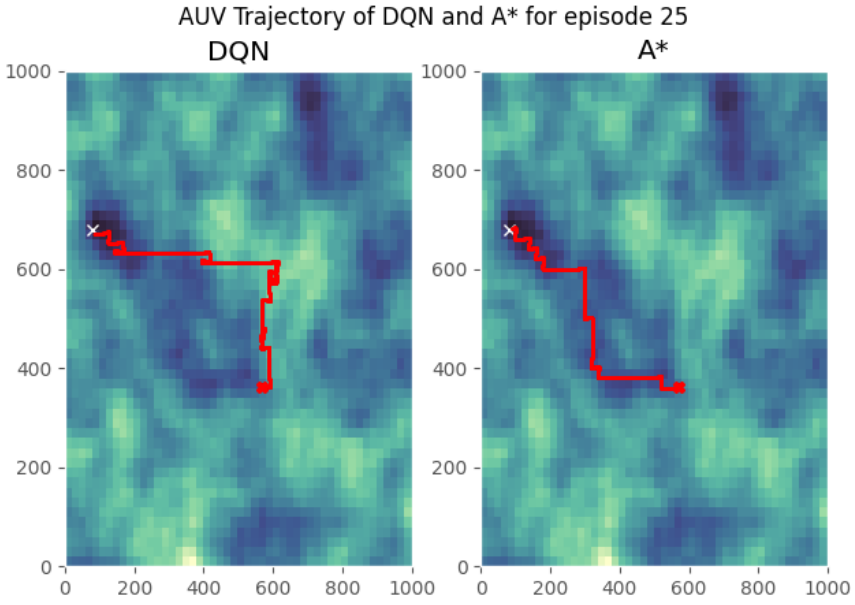

Figure A.26: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 25.

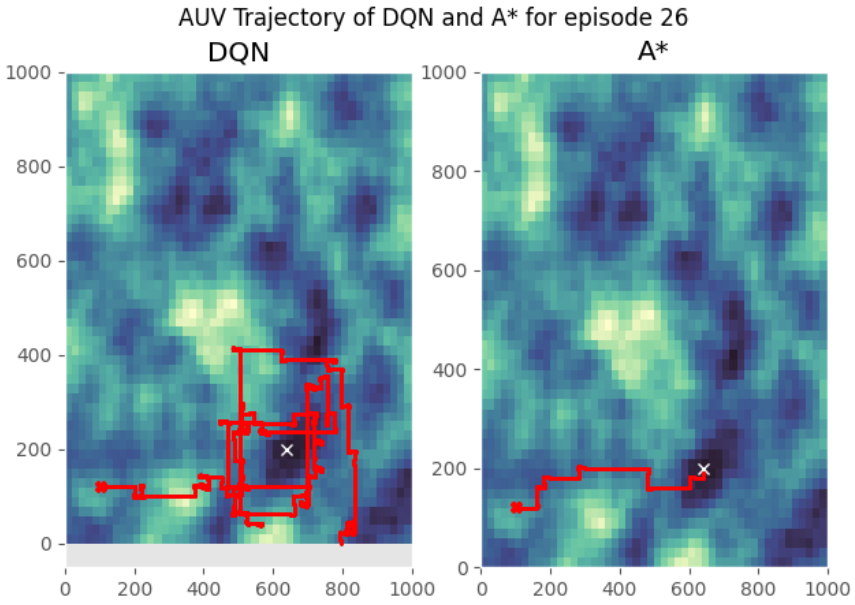

Figure A.27: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 26.

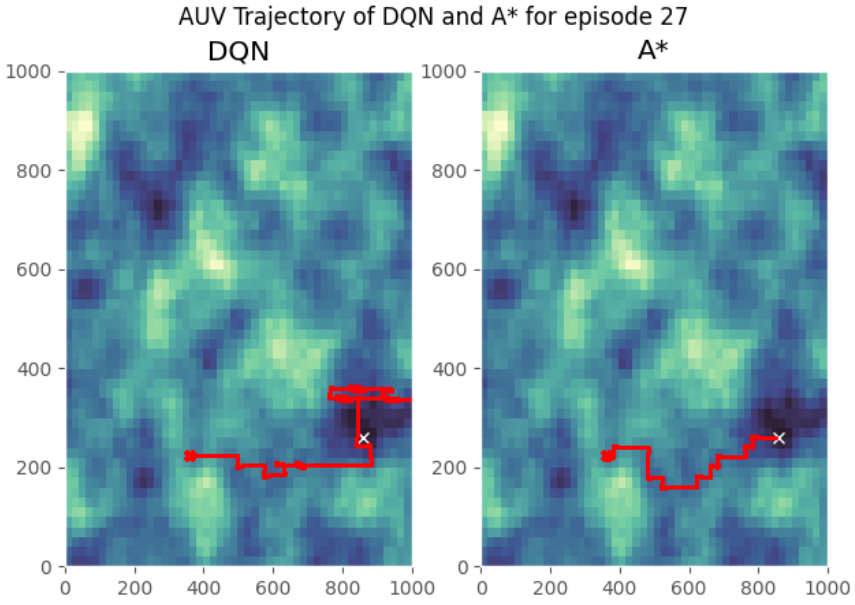

Figure A.28: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 27.

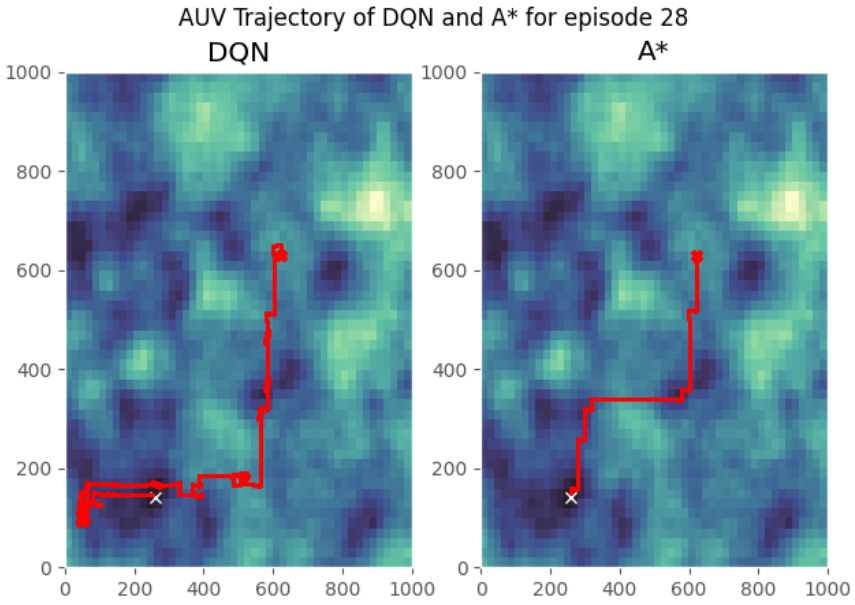

Figure A.29: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 28.

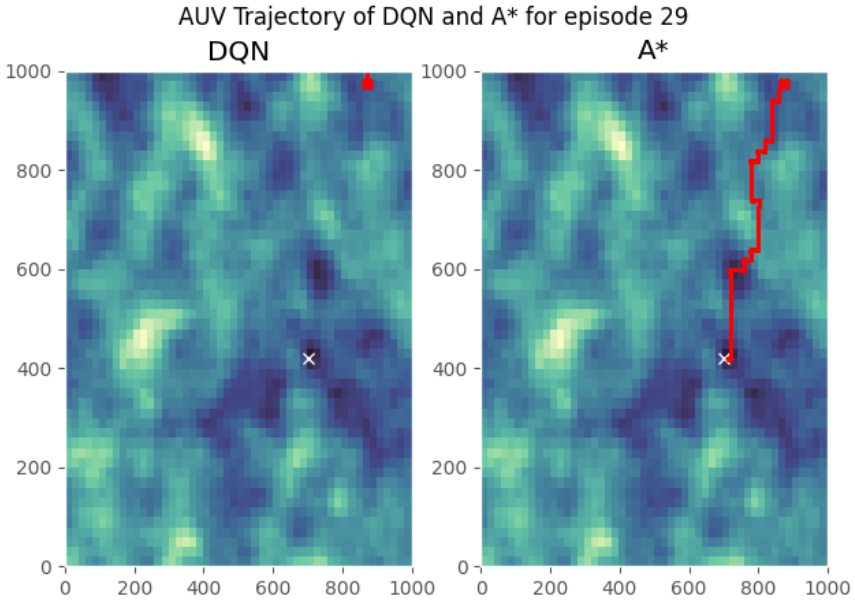

Figure A.30: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 29.

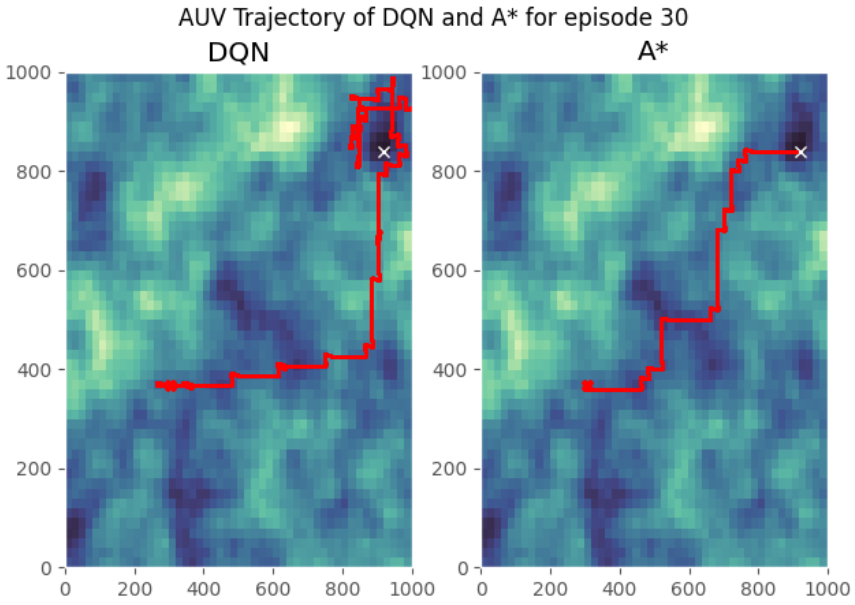

Figure A.31: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 30.

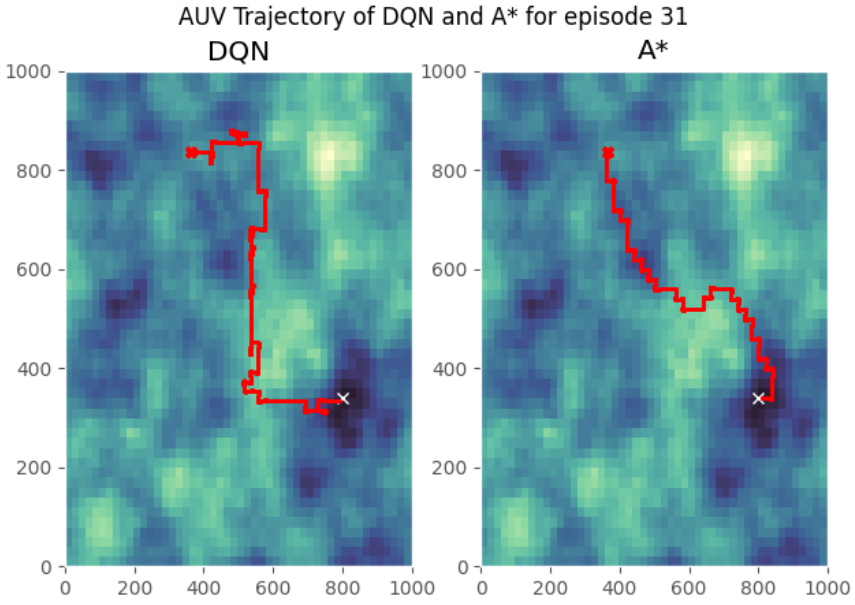

Figure A.32: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 31.

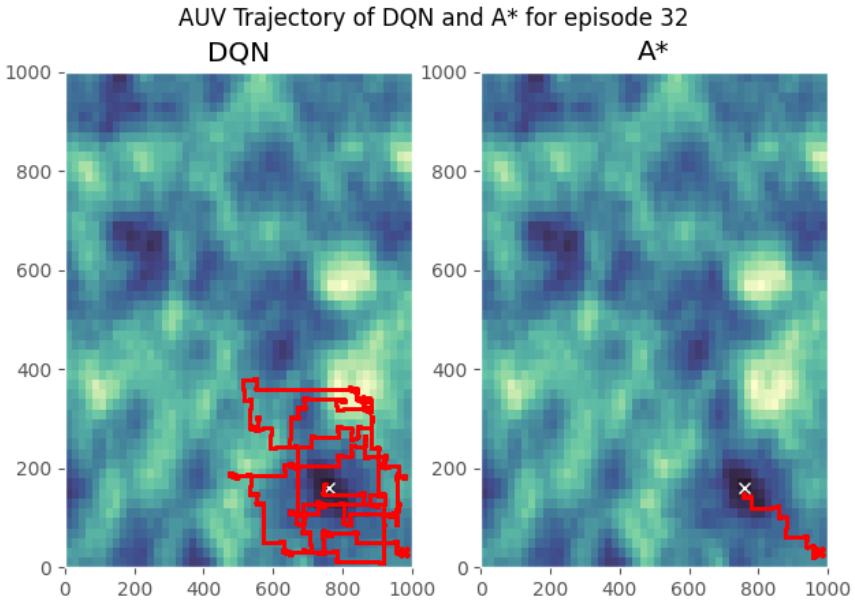

Figure A.33: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 32.

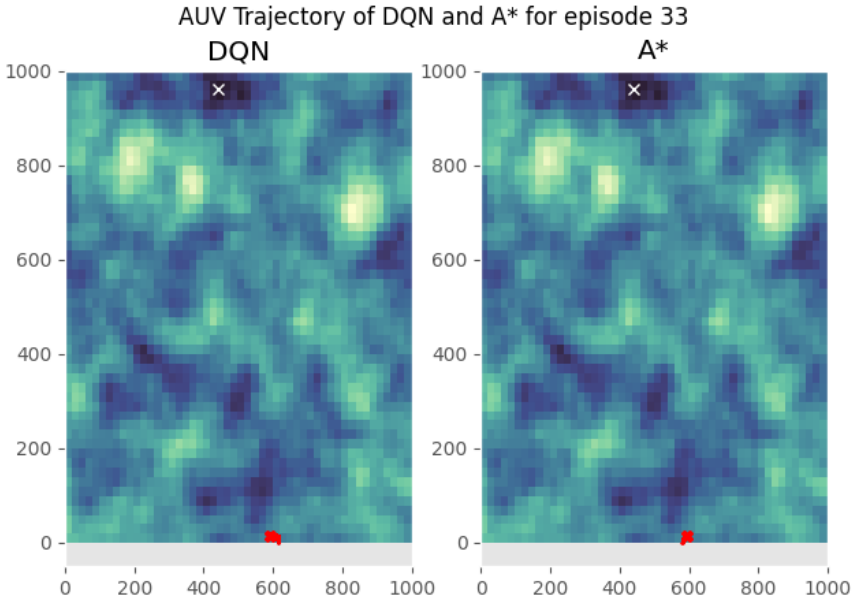

Figure A.34: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 33.

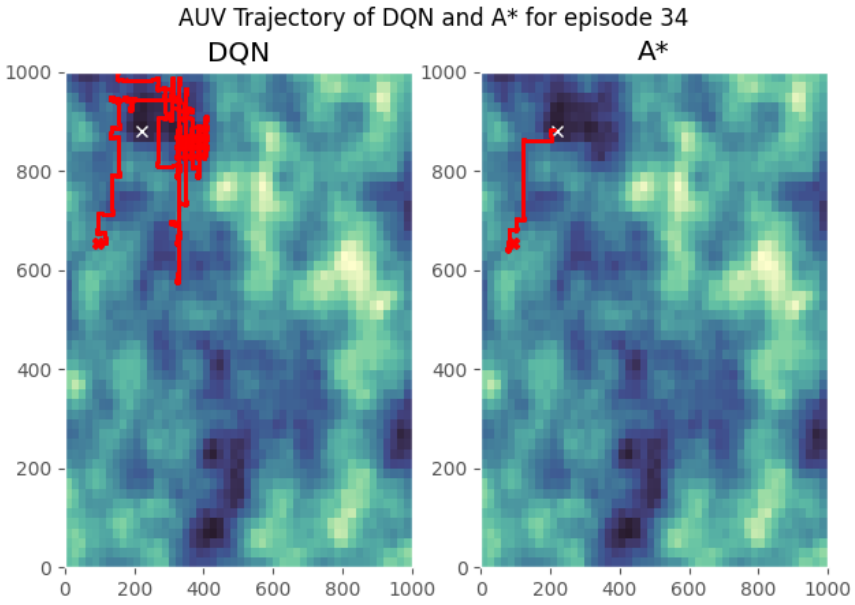

Figure A.35: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 34.

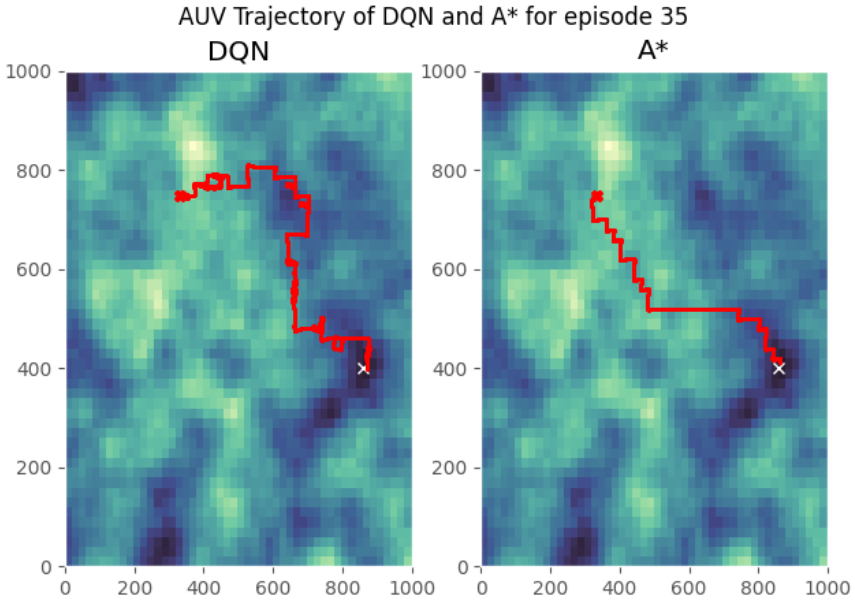

Figure A.36: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 35.

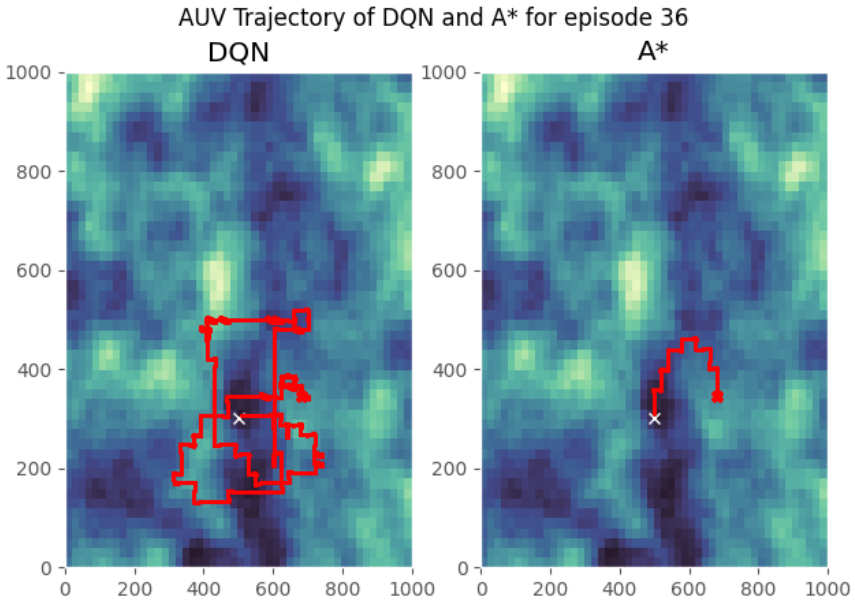

Figure A.37: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 36.

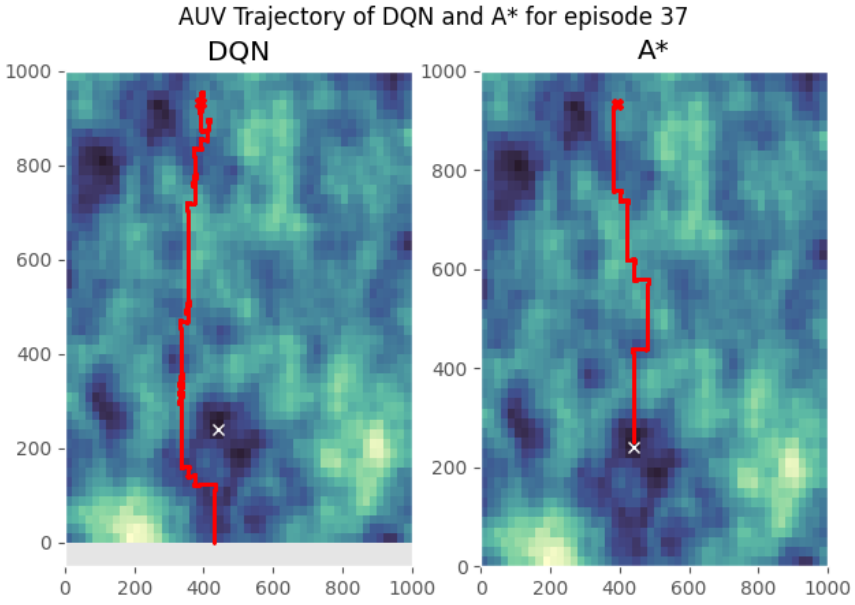

Figure A.38: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 37.

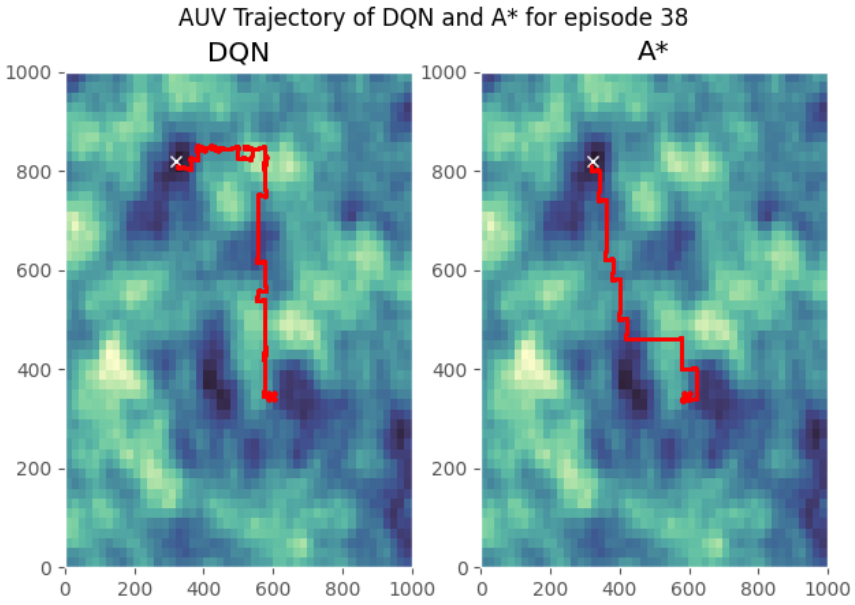

Figure A.39: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 38.

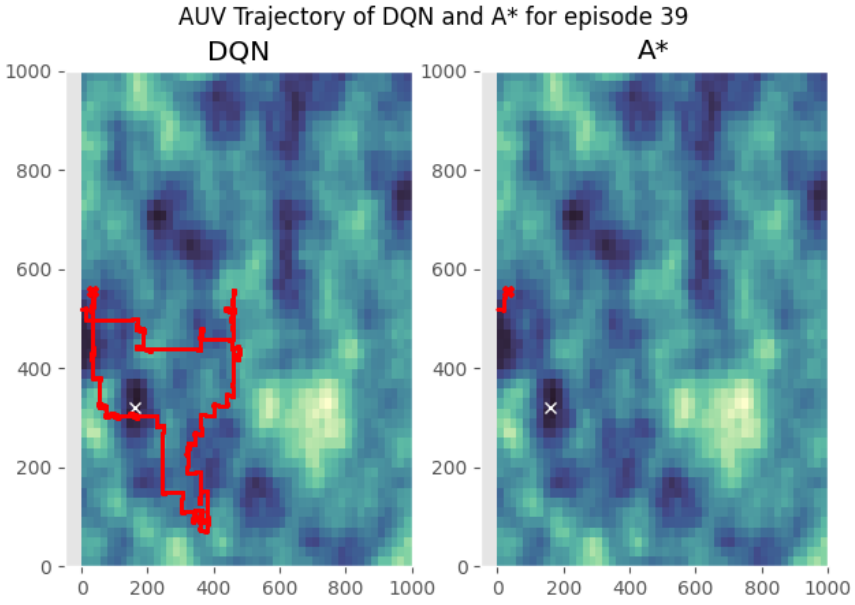

Figure A.40: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 39.

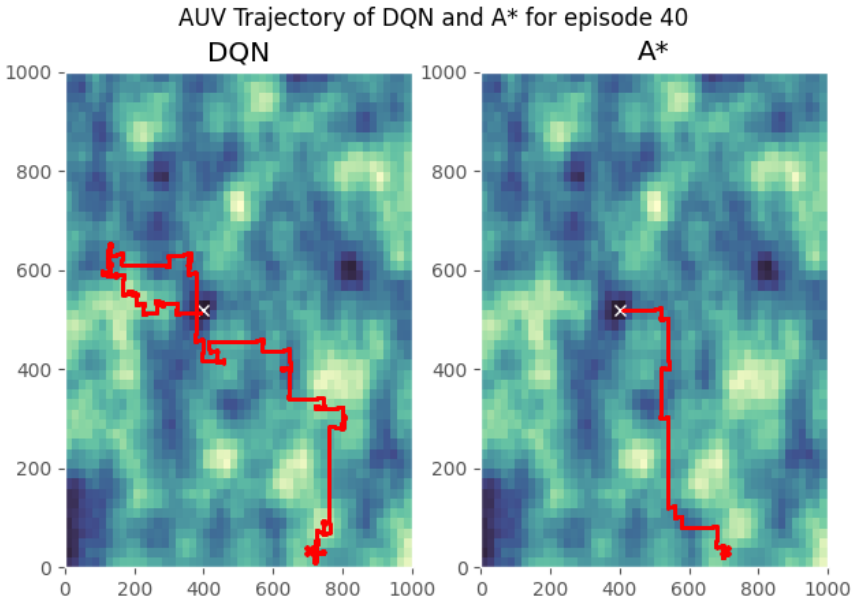

Figure A.41: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 40.

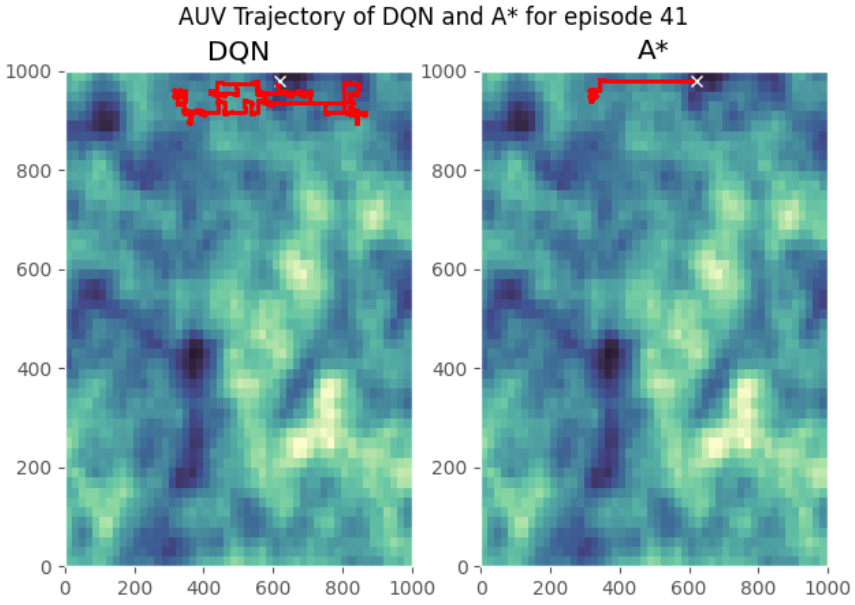

Figure A.42: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 41.

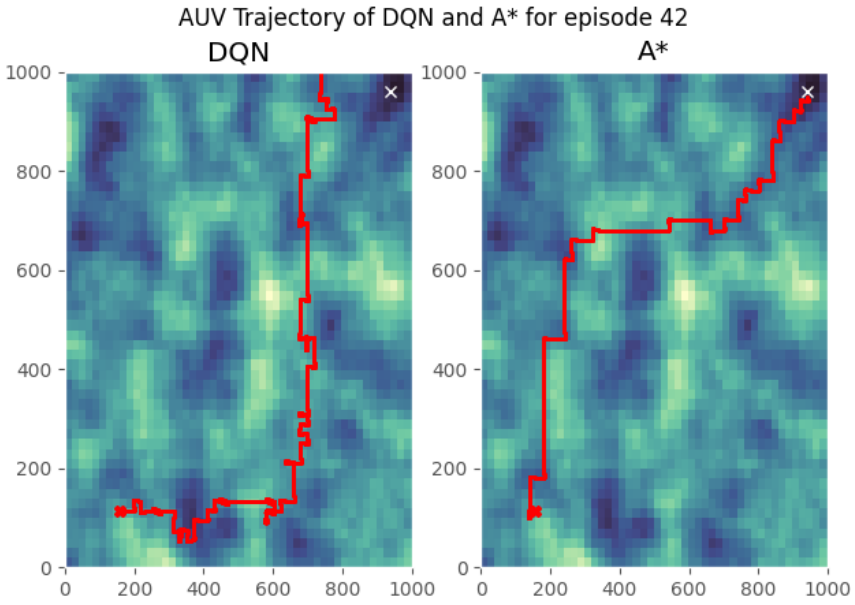

Figure A.43: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 42.

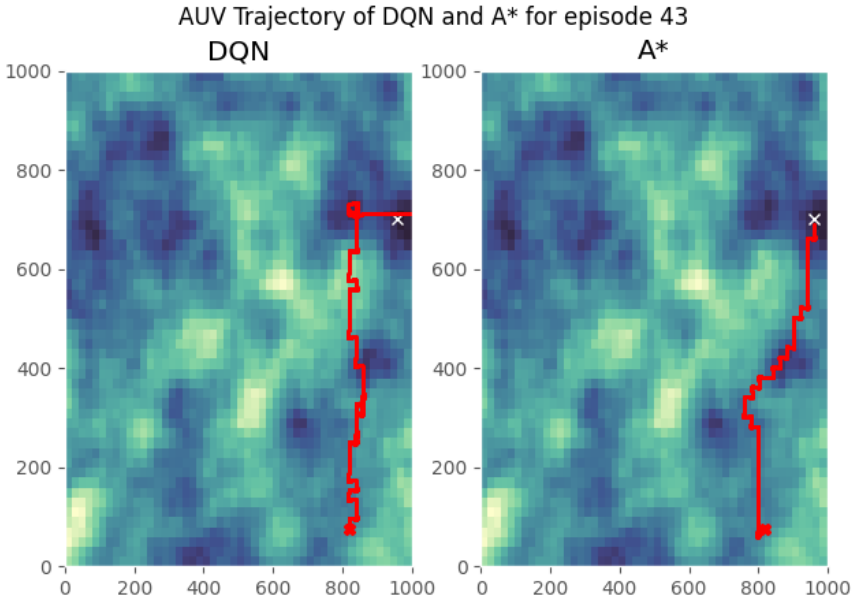

Figure A.44: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 43.

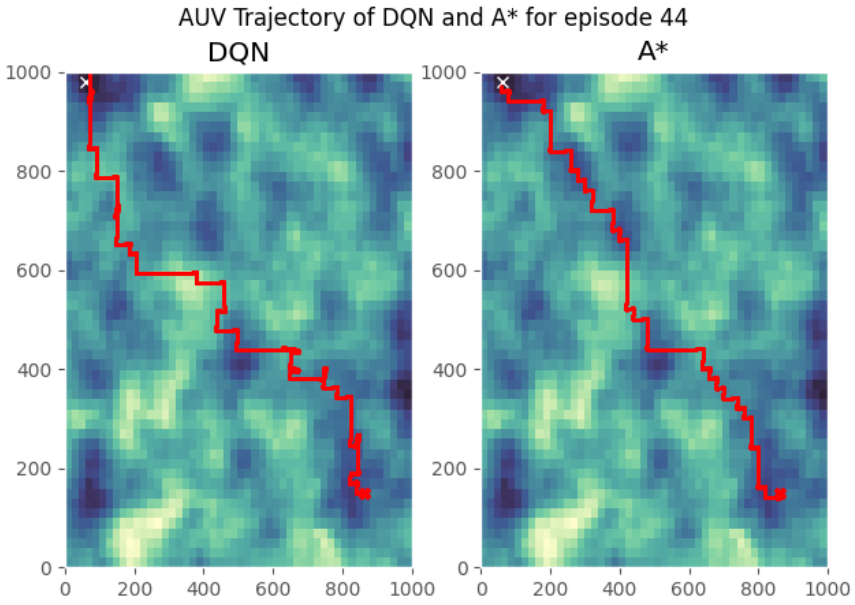

Figure A.45: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 44.

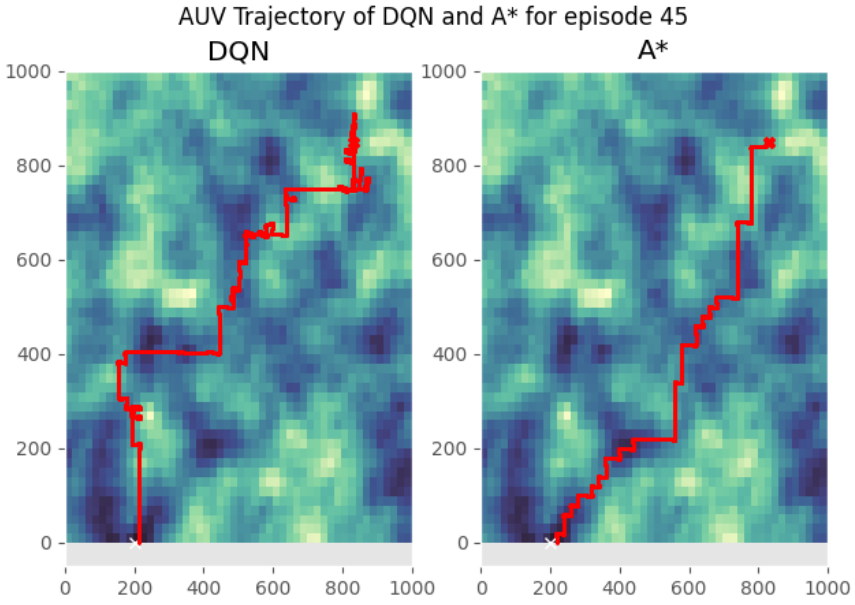

Figure A.46: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 45.

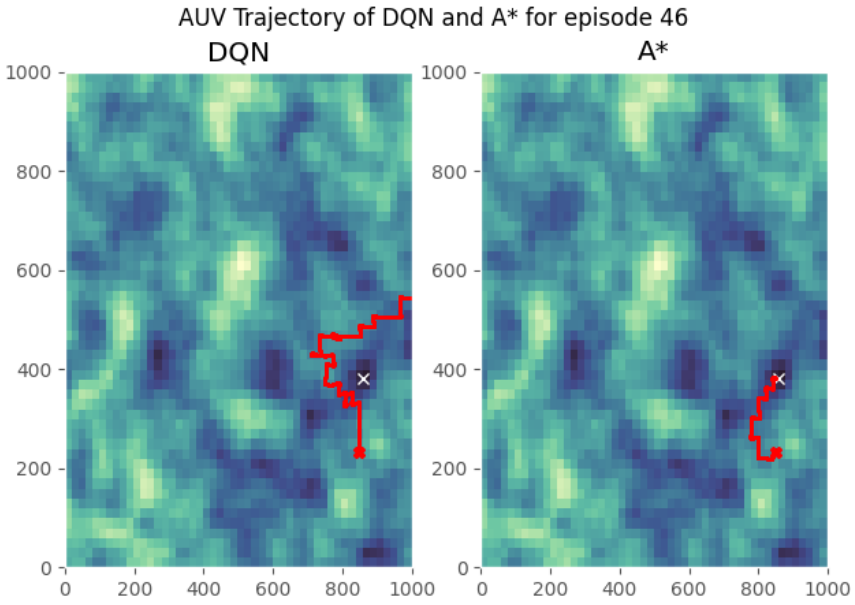

Figure A.47: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 46.

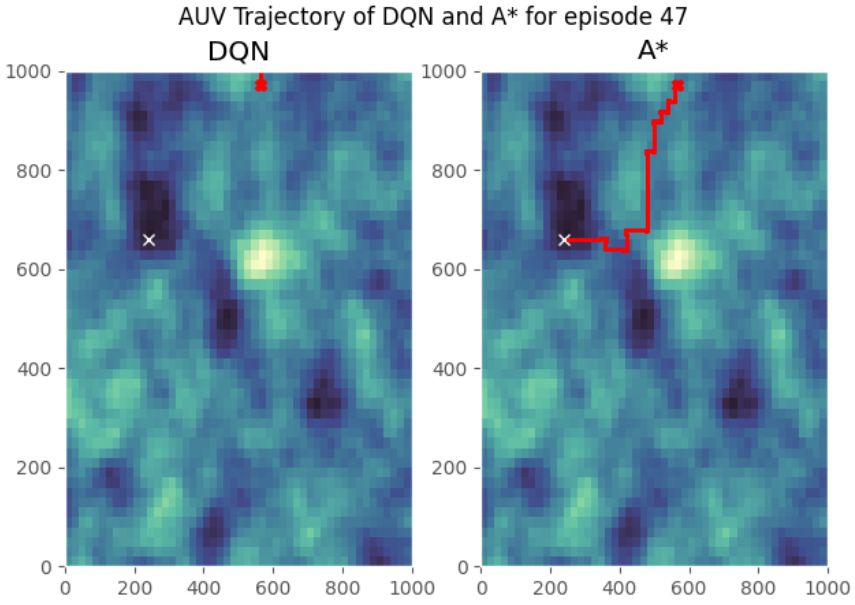

Figure A.48: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 47.

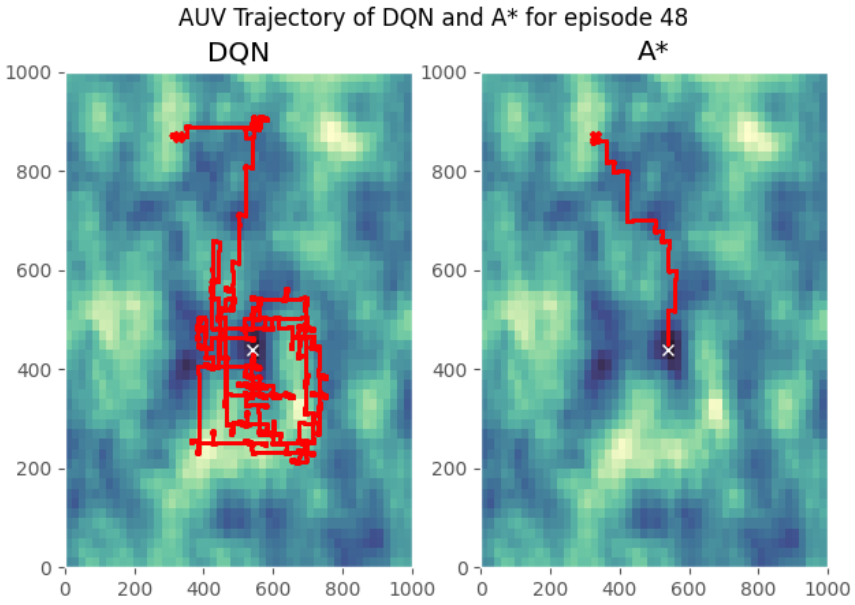

Figure A.49: A comparison of the trajectories made by the DQN agent and the A* algorithm for episode 48.

Ivan Håbjørg Kingman

DRL Applied to Targeted Oceanographic Sampling for an AUV

NTNU
Kunnskap for en bedre verden