Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Anmol Singh

Deep Reinforcement Learning for Spatio-Temporal Wildlife Management

Master's thesis in Computer Science
Supervisor: Keith L. Downing
June 2023

NTNU
Norwegian University of
Science and Technology

Anmol Singh

Deep Reinforcement Learning for Spatio-Temporal Wildlife Management

Master's thesis in Computer Science
Supervisor: Keith L. Downing
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Anmol Singh

Deep Reinforcement Learning for Spatio-Temporal Wildlife Management

Master's thesis, June 2023

Supervisor: Keith L. Downing

Data and Artificial Intelligence (DART)
Department of Computer Science (IDI)
Faculty of Information Technology and Electrical Engineering

Abstract

In the past 50 years, global wildlife populations have plummeted resulting in a biodiversity crisis where a significant number of species are at risk of extinction. This project tests the performance of various deep reinforcement learning (DRL) algorithms on the task of spatio-temporal wildlife management, for the purpose of maintaining biodiversity. DRL is a subfield of machine learning that combines deep neural networks with reinforcement learning, thus enabling an RL agent to solve complex problems in intricate environments.

In recent years, DRL has become increasingly popular due to its successful application in games like Chess and Atari 2600. However, there have been limited efforts to apply it to the field of wildlife management. Thus, the potential of DRL in this field remains largely unexplored. To address this, this thesis tests various DRL algorithms on the task of spatio-temporal wildlife management with the purpose of maintaining biodiversity. This was done by creating a spatio-temporal wildlife management simulation and training the DRL algorithms: DQN, A2C, and PPO on it. The focus of this thesis was to find the best action set for the RL agent, and the DRL algorithm with the best performance. The performance of the algorithms was based on the sizes of the species populations.

The results show that the best action set consists of actions that add fewer animals to the ecosystem. While all DRL algorithms were able to improve, the results indicate that there is a trade-off between performance and stability over training time. However, as training time increases PPO stands out as it performs similarly to DQN and A2C while being significantly more stable. This thesis shows the potential of DRL for wildlife management, and future work should investigate the applicability of other AI techniques, such as convolutional neural networks and evolutionary strategies, in this domain.

Sammendrag

(This is a Norwegian translation of the abstract)

De siste 50 årene har globale dyrelivsbestander opplevd betraktelig nedgang som har ført til en biologisk mangfoldskrise hvor et betydelig antall dyrearter har blitt utrydningstruet. Denne oppgaven tester ytelsen til ulike algoritmer av typen dyp forsterkende læring (DRL) innenfor romlig-temporær dyrelivsforvaltning, med formål om å opprettholde biologisk mangfold. DRL er et underområde av maskinlæring som kombinerer dype nevrale nettverk med forsterkende læring, slik at en RL agent kan løse komplekse problemer i avanserte miljøer.

I de siste årene har DRL blitt stadig mer populært grunnet sin vellykkede anvendelse i spill som sjakk og Atari 2600. Det har imidlertid vært få forsøk på å anvende det på dyrelivsforvaltning. Dermed er potensialet til DRL i dette feltet i stor grad uutforsket. For å adressere dette tester denne oppgaven ulike DRL-algoritmer innenfor romlig-temporær dyrelivsforvaltning, med formål om å opprettholde biologisk mangfold. Dette ble gjort ved å lage en romlig-temporær dyrelivsforvaltningssimulator, og trene DRL-algoritmene: DQN, A2C og PPO på den. Fokuset til denne oppgaven var å finne det beste handlingssettet for RL agenten, og å finne DRL-algoritmen med best ytelse. Ytelsen til algoritmene ble basert på størrelsene til artspopulasjonene.

Resultatene viser at det beste handlingssettet består av handlinger som legger til et mindre antall dyr til økosystemet. Selv om alle DRL-algoritmene forbedret seg, tyder resultatene på at det er en avveining mellom ytelse og stabilitet over treningstiden. Ettersom treningstiden øker, skiller PPO seg ut ved at den yter på nivå med DQN og A2C samtidig som den er betydelig mer stabil. Denne oppgaven har vist potensialet til DRL for dyrelivsforvaltning, og framtidig arbeid bør undersøke hvordan andre AI-teknikker, som konvolusjonelle nevrale nettverk og evolusjonsstrategier, kan anvendes i dette domenet.

Preface

This master's thesis was written at the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor Prof. Keith L. Downing for his valuable feedback and guidance during the course of this project.

Anmol Singh

Trondheim, 11th June 2023

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter serves as an introduction to this thesis. Section 1.1 covers the background and motivation behind the thesis. The research goal and research questions for this thesis are presented in section 1.2. Section 1.3 presents the research method used for answering the research questions. The contributions of this thesis are covered in section 6.3. Finally, section 1.5 gives an overview of the structure of this thesis.

1.1 Background and Motivation

The planet is currently facing a biodiversity crisis. According to the World Wildlife Fund (WWF), the global wildlife population has decreased by 69% in the last 50 years (WWF, 2022a). This is an unprecedented trend, that does not seem to slow down. There are several factors contributing to this trend, including changes in how humans use land and sea, habitat destruction, climate change, and invasive non-native species. Ecosystems are destroyed and altered as a consequence of these factors, which in turn lead to biodiversity loss.

To prevent ecosystems from reaching their tipping point and protect the world's wildlife, drastic measures are needed. In addition to more sustainable production and consumption, more wildlife conservation efforts are also needed. Wildlife conservation aims at protecting endangered wildlife and also facilitating sustainable ecosystems (WWF, 2022b). There are various ways of conserving wildlife, such as creating protected areas. This is a form of wildlife management. Wild-

life management aims at maintaining stable ecosystems through manipulation of population sizes (Mengak, 2008). This can be done through manipulating resources available for different species, but also through other measures, such as hunting. This thesis focuses on *manipulative* wildlife management, where the wildlife populations are altered directly. In this thesis, the term "manipulative" is often omitted, and "manipulative wildlife management" is simply written as "wildlife management".

In recent years, artificial intelligence (AI) has shown promise as a powerful tool for conservationists in various ways. AI technology has been used to identify and track wildlife, detect poachers and differentiate between species (Fritz, 2022). Much of this development is due to the increase in available computing hardware and available data. However, the AI technology used in these tools often relies on supervised learning, which in turn relies on large amounts of data. Since data collection can be expensive, and in some cases impractical, there is a need to explore new and creative ways for AI to facilitate wildlife conservation decisions. One such method, which has shown promise, is reinforcement learning (RL). RL is an AI technique that trains an agent to interact intelligently with an environment, through repeatedly taking actions and observing their effects on the environment. Since the agent learns solely through getting rewards by interacting with the environment, it is important to have a well-defined environment and a reasonable reward structure. It also means that there is no need for large amounts of data, which is an advantage RL has over supervised learning. Another advantage RL has in this field is that ecologists often create and utilize simulations, which can serve as a good starting base for an RL environment. Furthermore, deep neural networks can be combined with RL to solve increasingly complex problems. This is known as Deep Reinforcement Learning (DRL). These advances, in addition to some recent work done on RL in wildlife conservation (specifically the *CAPTAIN* framework), have shown promising results. To my knowledge, DRL has not yet been applied to the task of spatio-temporal wildlife management. This serves as the motivation for this thesis, where a spatio-temporal wildlife management simulation is created, and three DRL algorithms are trained and tested on it, with the aim of keeping a diverse and stable ecosystem.

1.2 Goal and Research Questions

The research goal and research questions of this thesis are stated below.

Goal Statement

> *To explore the use of different deep reinforcement learning (DRL) algorithms on the task of spatio-temporal wildlife management, with the aim of maintaining a diverse and stable ecosystem.*

Based on this goal statement, two research questions were formulated.

Research Question 1

> *What action set yields the best performance on the task of spatio-temporal wildlife management, when accounting for the costs that it entails?*

Research Question 2

> *Which DRL algorithm yields the best performance on the task of spatio-temporal wildlife management: Deep Q-Network (DQN), Advantage Actor-Critic (A2C), or Proximal Policy Optimization (PPO)? What are the trade-offs between these?*

Research question 1 looks to investigate what type of action enables the RL agent to keep a stable and diverse ecosystem when accounting for the costs that come with it. This research question is motivated by the fact that there is no obvious action set on the given task that enables the RL agents to act optimally and in a cost-effective way. The second research question is inspired by three popular DRL algorithms used in related work that have achieved valuable results in grid environment tasks: DQN, A2C, and PPO. The motivation behind this research question is to investigate which of these algorithms navigate the spatio-temporal wildlife environment best.

1.3 Research Method

To address the research goal and more specifically the research questions posed in this thesis, a spatio-temporal wildlife simulation was created. Such a simulation functions as an RL environment, where RL algorithms can and were tested. Another reason for creating such a simulation is that ecologists and conservationists regularly use simulation when taking decisions, hence it is not unfamiliar to the problem field. This makes it easier to find good models, which could be used in the simulation.

1.4 Contributions

To my knowledge, DRL has not been applied to spatio-temporal wildlife management before. Hence, this thesis contributes to the field of wildlife management by exploring the applicability of various DRL algorithms on it. This thesis gives a better understanding of how effective the different DRL algorithms are on the problem posed, which one yields the best performance, and what the most effective action set might be.

1.5 Thesis Structure

This thesis consists of 6 chapters. Chapter 1 serves as an introduction to the thesis, and presents the research goal and questions to be answered in this thesis. Background theory essential to understanding related work and methodology is presented in chapter 2. Related work in both the field of wildlife management and RL are covered in chapter 3. Chapter 4 presents the methodology used for answering the research questions, and how the experiments were carried out. The results from these experiments, as well as a discussion of them, are presented in chapter 5. Finally, in chapter 6, the research questions are answered and a conclusion is drawn.

Chapter 2

Background Theory and Motivation

The following chapter is based on a similar chapter written in my specialization project (Singh, 2022). This chapter aims to provide background knowledge on key concepts and techniques encountered later on in this thesis. Section 2.1 gives a brief overview of the field of wildlife management and traditional methods of conducting wildlife management. Section 2.2 introduces the mathematical basis for the wildlife simulation used in this thesis. Section 2.3 aims at providing a basic understanding of reinforcement learning to the reader. Section 2.4 serves as a summary of the whole chapter.

2.1 Wildlife Management

There are multiple different definitions and interpretations of wildlife management, Fryxell (2014) gives a general definition of the term as well as provides an overview of the field. According to Fryxell (2014, p. 2), wildlife management can be defined as "*management of wildlife populations in the context of the ecosystem*". The purpose of wildlife management has traditionally been *game*, hunting wildlife for sport, however in recent years aspects such as conservation and protection of endangered species have also been included.

There are two types of wildlife management, *custodial* and *manipulative*. Custodial management aims at protecting wildlife from external threats that might affect their population or habitat. Rather than directly influencing and stabilizing populations, custodial management focuses on letting natural processes be undisturbed. Manipulative management, on the other hand, regulates wildlife populations to keep the ecosystems in balance. This would be done by increasing the population if it is unacceptably low, or decreasing the population if it is unacceptably high. This kind of management is also used for harvesting purposes. The regulation of populations can either be direct, such as hunting and killing, or indirect, such as altering food supply and habitat. In this thesis, the RL agent will balance wildlife populations through manipulative management. For simplicity, the term "manipulative wildlife management" is simply referred to as "wildlife management" in this thesis.

Fryxell (2014) presents four different ways to manage wildlife population:

- Making it increase
- Making it decrease
- Harvesting it for a continuing yield
- Leaving it alone but observing it

What action to take depends on the current state of the ecosystem and the desired goal among other factors.

2.2 Simulating Biological Ecosystems

Biological ecosystems are complex, oftentimes consisting of multiple species interacting and affecting each other in an intricate environment. Species can have different relationships with each other, one of them being that of *predator-prey*. In a predator-prey relationship, the predator species consumes the prey species, which makes it possible for the predator species to survive and reproduce. The prey in that case might be a predator in another relationship with a species further down the food chain. A well-known and widely used way to model predator-prey ecosystems is through the Lotka-Volterra equations. These equations were developed by both the American biophysicist Alfred J. Lotka, and the Italian mathematician Vito Volterra independently of each other, at the same

time (Lotka, 1925; Volterra, 1928). The first-order differential equations are given below:

$$\frac{dN_1}{dt} = \alpha N_1 - \beta N_1 N_2 \tag{2.1}$$

$$\frac{dN_2}{dt} = \delta N_1 N_2 - \gamma N_2 \tag{2.2}$$

These equations model the growth over time of species 1, which is the prey, and species 2 which is the predator eating species 1. N_1 is the population of species 1, and N_2 is the population of species 2.

Equation 2.1 models the growth of the population of species 1 at a given time. The first term in the equation, αN_1, represents how the population of species 1 grows by consuming food (which is assumed to always be available to them). The parameter α represents the rate at which species 1 grows over a time step. The second term, $\beta N_1 N_2$, models how the population of species 1 decreases as a result of being eaten by species 2. This effect relies on how many predators there are, N_2, how many preys there are, N_1, and how many preys a predator eats over one time step, β.

Equation 2.2, similarly equation 2.1, models the growth in population for species 2. Just like the second term in equation 2.1, the first term in this equation represents the effect of species 2 hunting and eating species 1. However, instead of modeling the decrease in the population of species 1, this term models the increase in the population of species 2. The population of species 2 does not necessarily increase as much as the population of species 1 decreases. This is represented by having separate parameters for the rate of decrease in population for species 1, β, and the rate of increase in population for species 2, δ. The second term in the equation, γN_2, represents the decrease in the population of species 2 as a result of natural reasons such as death. The parameter γ represents the rate at which this decrease happens.

Most realistic predator-prey models, including the Lotka-Volterra model, have *cyclic dynamics*, meaning that there are cyclic fluctuations of both the prey and predator populations. It is worth noticing how the two different species' populations react to each other. If the prey population is high, the predator population is growing fast, but when the number of predators increases, more prey is eaten leading to their population decreasing. This would in turn lead to less food for the predators, meaning that their population would decrease. This cycle continues as long as the ecosystem remains undisturbed. Figure 2.1 shows

this cyclic dynamic through an example of predator and prey populations over time when modeled with Lotka-Volterra equations 2.1 and 2.2.

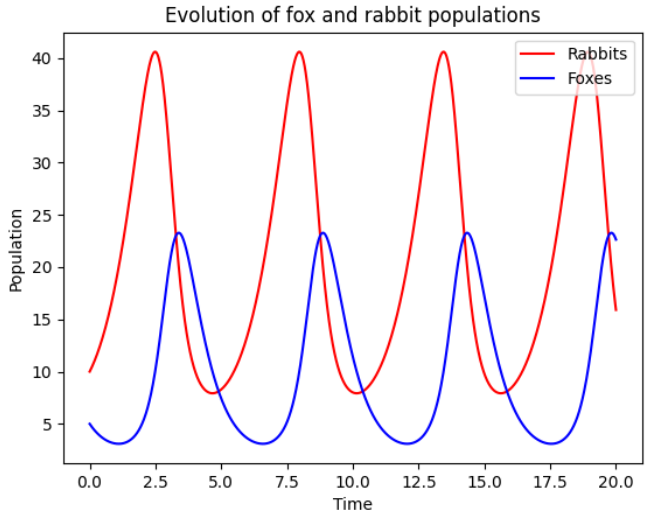


Figure 2.1: Graph over fox (predator) and rabbit (prey) populations over time in a Lotka-Volterra predator-prey model. Figure created using code provided by Bhupendra (2017).

Figure 2.1 shows the populations on the y-axis and time on the x-axis, but it is also possible to plot the two populations on their own separate axis to create a *phase-plane diagram*. Phase-plane diagrams can be useful to understand the dynamics of non-linear systems, such as the Lotka-Volterra model. Figure 2.2 shows the corresponding phase-plane diagram to figure 2.1. The phase-plot diagram shows a few of the cycles that the system can end up in. What cycle a system ends up in is decided by the initial conditions.

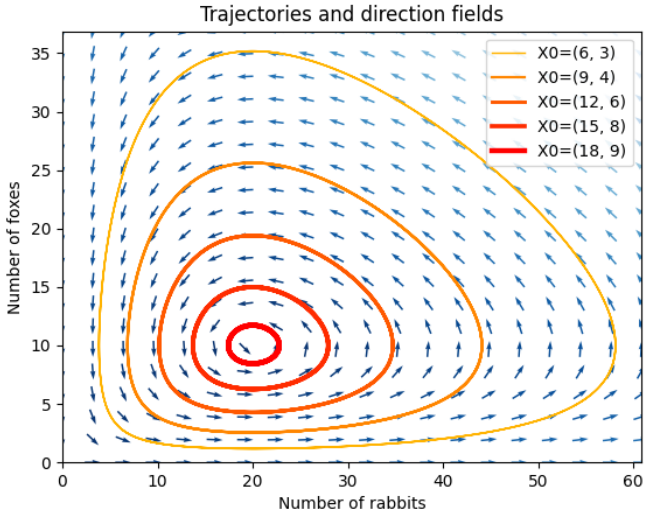


Figure 2.2: Phase-plane diagram over fox (predator) and rabbit (prey) populations in a Lotka-Volterra predator-prey model. The different cycles are determined by the initial conditions of the system. The smaller arrows show the direction that certain state develops towards. Figure created using code provided by Bhupendra (2017).

The Lotka-Volterra equations have over time been developed and modified into increasingly complex models that consider additional biological aspects. One such model is presented in Fryxell (2014, p. 158) and was based on work by Michael L. Rosenzweig and Robert H. MacArthur (Rosenzweig and MacArthur, 1963). The equations to this model are shown below:

$$\frac{dN_1}{dt} = r_{max}N_1(1 - \frac{N_1}{K}) - \frac{aN_1N_2}{1 + ahN_1} \tag{2.3}$$

$$\frac{dN_2}{dt} = \frac{acN_1N_2}{1 + ahN_1} - dN_2 \tag{2.4}$$

This model takes into account multiple biological factors, including *carrying capacity*. Carrying capacity refers to the average population size of a species an area is able to hold due to limited resources (Britannica, 2022). The original Lotka-Volterra equations have no limitation on the size of the prey population,

as shown by the first term in equation 2.1 which implies exponential growth. Rosenzweig and MacArthur's model incorporates carrying capacity mathematically by changing the first term in equation 2.1 from αN_1 to $r_{max}N_1(1 - \frac{N_1}{K})$ in equation 2.3, where r_{max} is the maximum reproduction per prey over a time unit and K is the carrying capacity. The first term changes from implying exponential growth to logistic growth, which is more realistic as there is always limited resources in ecosystems.

Another biological aspect considered by this model is *functional response*. Functional response is a term popularized by the Canadian ecologist C. S. Holling, and it refers to the individual predator's behavior when it comes to consuming prey (Holling, 1959). Holling described three types of functional responses. *Type 1* functional response assumes that the predator has an unlimited appetite, meaning that the number of prey it consumes grows linearly with the prey density. This type of behavior is assumed in the Lotka-Volterra equations but is an unrealistic assumption as no animal has an unlimited appetite. *Type 2* functional response has the more realistic assumption of predators having a limited appetite. This means that the number of prey eaten by a predator levels off as the prey density grows. *Type 3* functional response is similar to type 2 but additionally takes into account that it is more difficult for predators to find and consume prey at lower prey densities. Rosenzweig and MacArthur's model adopts the type 2 functional response. This type of response is represented by the second term of equation 2.3 and the first term of equation 2.4. The parameter a represents the amount of area predators search during one time unit, and h represents the *handling time*, the time it takes for a predator to consume one prey. Conversion of prey consumption to predator offspring is represented by the parameter c. Equation 2.4 also contains a parameter d, which is similar to γ in equation 2.2 in that they represent the decrease in the predator population due to natural reasons such as death, over one time unit.

The dynamics of Rosenzweig and MacArthur's model can vary, and depend on the values of the parameters in the model. Two of the possible dynamics are explained by Fryxell (2014): *stable limit cycle* and *stable equilibrium*. A model has a stable limit cycle if all starting states converge towards a pattern of repeating cycles, this is shown in figure 2.3. Depending on the carrying capacity of the prey, the system can also converge toward an equilibrium, where both the prey and predator populations are constant. This is shown in figure 2.4. A more detailed analysis of the different dynamics was carried out by Tanner (1975).

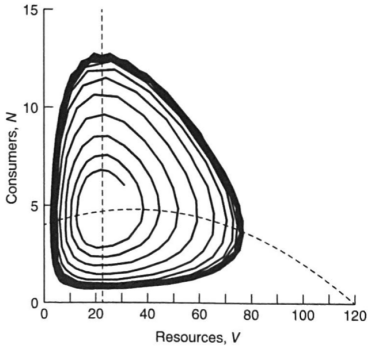


Figure 2.3: Phase-plane diagram of a stable limit cycle (Fryxell, 2014).

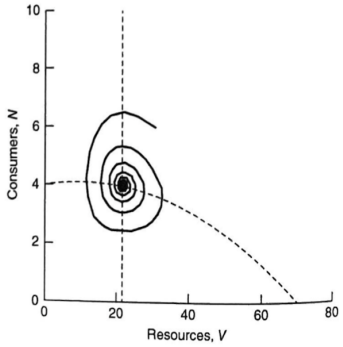


Figure 2.4: Phase-plane diagram of a stable equilibrium (Fryxell, 2014).

Rosenzweig and MacArthur's two-species predator-prey model can be expanded into a three-species model. There are multiple different relationships that three species can have in nature, such as two predator species competing for one prey species or one predator species consuming two prey species. Another type of relationship can be found in tri-trophic systems, where there is one predator species (hereby referred to as the apex predator) consuming another predator species (hereby referred to as the mesopredator), which in turn consumes the prey. This kind of tri-trophic model is described in Fryxell (2014, p. 167), and its equations are given below:

$$\frac{dN_1}{dt} = r_{max}(1 - \frac{N_1}{K}) - \frac{aN_1N_2}{b + N_1} \tag{2.5}$$

$$\frac{dN_2}{dt} = \frac{aeN_1N_2}{b + N_1} - dN_2 - \frac{AN_2N_3}{B + N_2} \tag{2.6}$$

$$\frac{dN_3}{dt} = \frac{AEN_2N_3}{B + N_2} - DN_3 \tag{2.7}$$

This model shares many of the parameters with the two-species model described in the equations 2.3 and 2.4, however, there are some differences and additional parameters. The parameter a represents the rate of prey consumption by a mesopredator, and b represents the number at which the mesopredator consumption of

the prey is half of its maximum. Parameter e represents the same as c in equation 2.4. The uppercase parameters represent the same as the lowercase parameters but with respect to the relationship between the mesopredators and the apex predators. Another difference is that the type 2 functional response is modeled with the Michaelis-Menten equation[1].

This type of tri-trophic model has been analyzed in Hastings and Powell (1991), where it is shown that the different populations have complex aperiodic fluctuations that never repeat, meaning that there is deterministic chaos. A phase-plane diagram of the system is shown in figure 2.5.

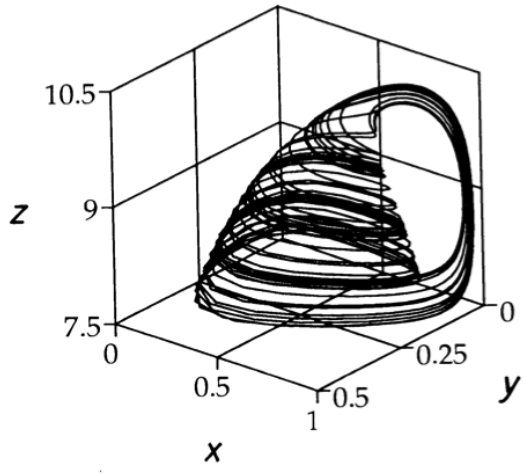


Figure 2.5: Phase-plane diagram of a tri-trophic model, showing its dynamics. The x-axis represents the prey population, the y-axis represents the mesopredator population, and the z-axis represents the apex predator population (Hastings and Powell, 1991). The diagram resembles an upside-down teacup. The prey and mesopredator populations fluctuate while the apex predator population is increasing, creating the "cup", until the apex predator population collapses, creating the "handle".

[1]The Michaelis-Menten equation stems from biochemistry and models enzyme kinetics. The equations can be used to model type 2 functional response as the reaction rate levels off with growing substrate concentration. The equation contains a variable (b in equation 2.6 and B in equation 2.7) which represents the point at which the reaction rate is half of its maximum. In a predator-prey system, reaction rate would represent the rate at which a predator consumes prey, and substrate concentration would represent prey density (Hauge, 2020).

To stabilize a chaotic tri-trophic system, it is possible to introduce intraspecific competition at the apex predator level, meaning competition between individuals of the same species. This is not uncommon in nature, one example being wolf packs competing with each other for territory. To incorporate this into the tri-trophic model, equation 2.7 can be replaced with equation 2.8.

$$\frac{dN_3}{dt} = \frac{AEN_2N_3}{B + N_2} - DN_3 - \frac{sN_3^2}{\gamma} \tag{2.8}$$

The parameter s is the maximum rate of apex predators per capita, and γ is the maximum apex predator density. This additional mortality rate stabilizes the system from chaos to a stable limit cycle.

The models discussed represent a simplified and idealized representation of the real world, hence there are some unrealistic scenarios that might arise when working with these models. One of these scenarios is named the *atto-fox problem* (Lobry and Sari, 2015). The atto-fox problem refers to systems displaying a stable limit cycle where populations become unrealistically low but still manage to rebound and grow. For example, the predator population could shrink to 10^{-18} individuals, and still be able to grow. This is of course not possible in the real world, as any species with less than two individuals would become extinct. One simple approach to handle this problem is to set a population threshold for each species for when they are declared extinct.

In summary, simple predator-prey models can become increasingly complex when taking into account realistic biological aspects. When going from a two-species model to a tri-trophic system, the model can even display chaos.

2.3 Reinforcement Learning

Reinforcement learning (RL) is a field of machine learning that focuses on training agents to behave intelligently in an environment through getting rewards. While supervised learning relies on labeled data to learn, RL only needs a defined environment and reward function to learn how to act intelligently in the environment.

2.3.1 General RL System

An overview of a general RL system is shown in figure 2.6. An agent performs an action that changes the environment. The agent then observes the new state it is in, and additionally gets rewards (also called reinforcements) based on this state.

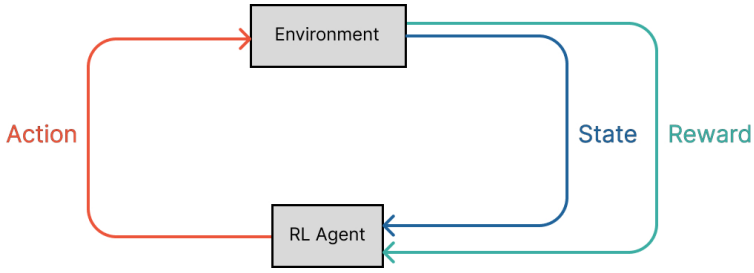


Figure 2.6: An overview of a general RL system. An agent interacts with an environment by taking actions, moving to new states, and receiving rewards based on the state that it ends up in.

The environment that an RL agent interacts with can be viewed as a *Markov Decision process* (MDP). Russell (2016, p. 647) gave the following definition of an MDP:

> *"A sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards ..., and consists of a set of states (with an initial state s_0); a set $ACTIONS(s)$ of actions in each state; a transition model $P(s'|s,a)$; and a reward function $R(s)$."*

A Markovian transition model relies on the *Markov assumption*: The current state is only dependent on a fixed finite number of previous states. Usually, a first-order Markov process is assumed, meaning that the current state only depends on the previous state. The Markov assumption simplifies the transition model, as it doesn't need to take into account the full history of states the agent has been in when predicting the next state.

Based on the actions the agent takes, and the rewards it receives, the agent constructs a *policy* (often denoted by π). A policy tells the agent what action to take in a certain state. An optimal policy will suggest the action that leads

to the highest expected value (also referred to as utility). One way to define the value of a state is through the Bellman equation (Sutton and Barto, 2018):

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.9}$$

The equation defines the value of being in a state s given policy π, as the immediate reward, r, of getting to a state s', plus the discounted (γ is the discount factor for future rewards) expected value of that next state s' (given the policy π), all of which is weighted by the probability of getting that reward and ending up in that state, given that action a is performed in state s. This calculation is done for all possible states s' and rewards r, given action a is performed in state s, and summed over. This sum is weighted by the probability of performing action a in state s, given the policy π. This weighted sum is summed over for all possible actions a. In short, the Bellman equation defines the value of a state s, given a policy π, as the sum of all possible future values and rewards (s' and r), weighted by their probability.

The term $p(s', r|s, a)$ in equation 2.9 is known as the *transition model* of the environment. When the reward function is deterministic, meaning that the reward associated with going from any state s to s', when performing action a, is deterministic, this term can be simplified to $p(s'|s, a)$. In this thesis the RL problem will have a deterministic reward function, hence the rest of this chapter will utilize this simplification.

RL agents can either be *model-based* or *model-free*. A model-based agent aims to learn the transition model of the environment. The transition model of the environment can be learned by keeping track of the number of times the agent ends up in state s' when performing action a in state s, and dividing that by the total number of times the agent has performed action a in state s. The learned transition model can be inserted into a variation of the Bellman equation, and the value function could then be solved by dynamic programming. The value function could then be used to decide which action in a certain state would give the highest expected utility. However, dynamic programming is limited in that it is computationally infeasible when the state space becomes very large.

2.3.2 TD-learning

A model-free agent does not require a model of the environment. One such approach is *Temporal-difference learning* (TD learning). TD learning updates the value of a state S under a policy π with the value of a successive state S', hence the name temporal-difference (Sutton and Barto, 2018). The value update equation for *one-step* TD, also called *TD(0)*, is given below:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.10}$$

Here, V represents an estimate of the value function v_π, α represents the learning rate: how much influence the observation (the transition from S_t to S_{t+1}) should have on the value of S_t. The reward received when going from S_t to S_{t+1}, is represented by R_{t+1}. The component $R_{t+1} + \gamma V(S_{t+1})$ is referred to as the *TD-target*, while $R_{t+1}+\gamma V(S_{t+1})-V(S_t)$ is the *TD-error*. Since TD-learning updates state-values using other existing state-value estimates, it is a *bootstrapping* method.

Equation 2.10 is known as one-step TD as the estimated value of the first successive state, $V(S_{t+1})$, is used. If the value of the n-th successive state had been used it would be called n-step TD, and the TD-target would be:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{2.11}$$

$G_{t:t+n}$ is known as the n-step return. V_{t+n-1} is the estimate of v_π at time $t+n-1$. If n goes to infinity (meaning to a terminal state in the environment), it would be called a *Monte Carlo* method. In Monte Carlo methods the TD-target is the sum of discounted rewards only, and does not use any state-value estimate, hence it is not a bootstrapping method.

The idea behind TD learning is that the value of a state will converge toward its true equilibrium as the number of observations increases. This convergence may take more time, but in exchange, the agent does not need to learn a model of the environment.

Both dynamic programming and TD learning approaches calculate the values of the different states in the environment, under a certain policy. However, for most applications, it would be useful if the agent itself learned what actions to take. To do this the agent needs to *explore* the different actions it can take in different states, in order to gain knowledge about the environment. However, it also needs to *exploit* the knowledge it gains by taking actions that lead to high-value states.

The balance between exploration and exploitation is important as it affects the agent's ability to reach a reasonably good policy. One way to balance these two aspects is to assign a large positive value estimate to relatively undiscovered state-action pairs, which decreases with the number of times that state-action pair is explored.

2.3.3 Q-learning and SARSA

One notable RL method is *Q-learning*. Q-learning is a one-step TD learning approach, that instead of state-values, learns *Q-values*. Q-values can be viewed as the value of executing an action a in a state s. The update equation for Q-values is similar to the one for state-values in TD-learning (equation 2.10):

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{2.12}$$

While equation 2.10 calculated values based on a fixed policy π, equation 2.12 does not. The Q-value $Q(s,a)$ is calculated based on the highest Q-value in the successive state, $\max_{a'} Q(s',a')$. It is important to note that the selection of action to execute in the successive state s' is not based on what results in the highest Q-value. The selection of action to take in a state needs to take into account the balance between exploration and exploitation, and might therefore not choose the action leading to the highest Q-value. The algorithm chooses actions to execute according to a *behavior policy* while constructing a *target policy* aimed to approximate the optimal one. Because of this, Q-learning is referred to as an *off-policy*, as the Q-values are not calculated based on the actual policy of the agent (the behavior policy), but on the action a' that results in the highest $Q(s',a')$ (the target policy). An *on-policy* method, such as *SARSA* (State-Action-Reward-State-Action), would use the executed action a' in the successive state s' when calculating the Q-value $Q(s,a)$, meaning that the target policy equals the behavior policy. The equation for updating Q-values in a SARSA approach is given below:

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a)) \tag{2.13}$$

The distinction between Q-learning and SARSA can be seen by the absence of the $\max_{a'}$ operator. The action a' in the equation is the one that is actually executed by the agent in the successive state s'.

To decide what actions to execute when performing Q-learning (or SARSA), it is usual to utilize an ϵ-*greedy* policy. An ϵ-greedy policy usually chooses the action with the highest Q-value, but, with a probability of ϵ, it might choose an action randomly. This allows for both exploration and exploitation of what the agent has learned.

One potential concern of Q-learning is *maximation bias*. Maximation bias refers to the positive bias that might occur when choosing the action leading to the maximum Q-value. The positive bias might occur since the maximization is done over an estimate of the true Q-value, and not the actual true Q-values. This bias can in turn lead to non-optimal behavior for the RL agent, which in turn would affect its performance.

The *Deep Q-Network (DQN)* algorithm combines Q-learning and neural networks, which are described in the next section. The DQN algorithm is described in further detail in section 3.4.1.

2.3.4 Neural Networks as Function Approximators

An important aspect of RL algorithms is how to represent the value function (or Q-function). It is possible to store these in a table, one entry for each state (or state-action pair), together with its value (or Q-value). However, when the state space becomes large, this becomes infeasible. Another issue with this approach is that the agent does not have any information about the value of a state it never has been in, as it does not have an entry for that state in the table.

These issues can be handled by function approximation. Function approximation aims to learn an approximation of the true value function. *Neural networks*, also called *Artificial Neural Networks*, is one way to do this. Neural networks, more precisely, multi-layer perceptrons, are non-linear universal function approximators (Hornik et al., 1989). This means that they have the capacity to approximate any function, and this can be done with significantly fewer parameters than states in an environment.

Another valuable aspect of function approximation is that it allows for generalization, meaning that by learning about the value of one state, the function approximator could also learn about the values of other states, even the ones the agent has never visited. This is because the function approximator changes its parameters based on the value it observes of a state at a certain time, and the parameters might also change the function approximator's value estimates of other states.

These attributes, in addition to recent advances in deep learning, have made deep RL (DRL) increasingly popular. Deep learning refers to the use of deep neural networks, meaning networks with multiple hidden layers. However, the use of neural networks is not problem free. The networks navigate through the solution space for the approximate function and might end up in local minima, which could restrict them from finding good solutions.

2.3.5 Actor-Critic

The RL methods discussed in previous sections focused on learning a good value function (e.g. Q-values) and choosing actions based on this. An alternative approach is to learn both a value function and a policy function. The policy function chooses what actions to take. This approach is called *actor-critic* methods, where the *actor* refers to the policy that is learned, and the *critic* refers to the learned value function.

Actor-critic methods fall under a category of RL methods called *policy gradient methods*. Policy gradient methods aim to learn a parameterized policy to take actions, rather than basing what actions to take on a learned value function. However, a value function might still aid in learning the parameters of the policy, such as in actor-critic methods. The objective of policy gradient methods is to maximize some performance measurement, $J(\theta)$, where θ represents the parameters of the policy. If the performance measure is defined as the true value of the starting state given the policy π_θ determined by a set of parameters θ, the *policy gradient theorem* (PGT) can be proven (Sutton and Barto, 2018, p. 326):

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a Q_\pi(s,a) \nabla_\theta \pi(a|s,\theta) \tag{2.14}$$

which says that the partial derivative of the performance measurement with respect to the policy parameters, $\nabla_\theta J(\theta)$, is proportional to the sum of the probability of being in a state s given policy π, $\mu(s)$, times the sum, over all possible actions a, of the Q-values for that action-state pair, which is multiplied with the partial derivative of the probability, under policy π, of choosing that action given the state and policy parameters. PGT presents an expression for the gradient of the performance with regard to the parameters of the policy. Policy gradient methods aim to make changes to the policy parameters θ that are close to these gradients.

The classical REINFORCE algorithm updates its policy parameters by the following equation:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \tag{2.15}$$

where the next time step policy parameters θ_{t+1} are equal to the current time step policy parameters θ_t, plus a term in the direction of the partial derivative of the policy. This term is the product of the step size α, the cumulative discounted reward from time step t till the end of the episode G_t and the partial derivative of the policy probability of selecting action A_t in state S_t with regards to the policy parameters, divided by the policy probability of that action, given the same state.

As REINFORCE uses the accumulated rewards until the episode end to update its policy, it is a Monte Carlo approach. Actor-critic methods, on the other hand, update policy parameters (and value function parameters) using TD error:

$$\theta_{t+1} = \theta_t + \alpha \delta_t \frac{\nabla_\theta \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \tag{2.16}$$

where δ_t is the TD error at time step t:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$$

This equation is similar to the TD error in equation 2.10. $\hat{v}$ is the state value function parameterized by w, and is also referred to as the critic. Actor-critic methods are considered TD methods as the state-values are used to update the policy indirectly through the TD error.

The term $\frac{\nabla_\theta \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}$ in equation 2.16 is often written as $\nabla_\theta \log \pi(A_t|S_t, \theta_t)$ in literature and pseudocode. This is based on the identity $\nabla \log x = \frac{\nabla x}{x}$. It is also common to refer to the TD error as an estimator for the *advantage function*. The advantage function measures the *advantage* of taking an action in a state and is therefore formally defined as $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$. Notice that the notation for the action taken at time t has changed from A_t to a_t, as the capital A is now representing the advantage function. By employing this notation, the equation for the policy gradient estimator which is most frequently used, is obtained:

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a|s_t)\hat{A}_t] \tag{2.17}$$

where $\hat{g}$ is the policy gradient estimator, $\hat{\mathbb{E}}_t$ represents the average over a finite batch of experiences, $\pi_\theta(a|s_t)$ is the policy given the parameters θ, and $\hat{A}_t$ is the advantage function at time t.

An overview of the general actor-critic architecture is shown in figure 2.7.

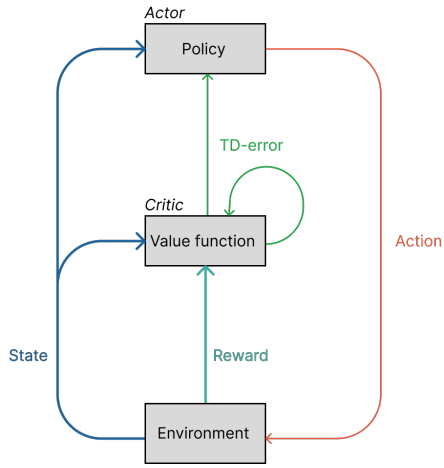


Figure 2.7: An overview of general actor-critic architecture. Both the actor and critic receive the state of the environment. The critic also receives the reward related to the previous state-action pair. The critic uses the reward and the new state to calculate the TD error, which is used to calculate its loss gradients. The TD error is sent to the actor to update its policy parameters according to equation 2.16. The actor also chooses the next action, based on its parameters and the new environment state.

Parameterizing policy might have several advantages over value parameterization, with one being that a good policy function might be easier to approximate than a good value function. This is naturally dependent on the RL task and might vary from problem to problem. Another advantage of approximating policy function is that the action selection probabilities change more smoothly than an ϵ-greedy strategy based on a value function would. This is because a small change in action-values could change what action has the highest value, thereby significantly changing the action selection probabilities. Since the policy

function changes more smoothly in policy-gradient methods, it has stronger convergence guarantees (Sutton and Barto, 2018).

The actor-critic algorithms *A2C* and *PPO*, which are tested on the spatio-temporal simulation in this thesis, are presented in section 3.4.2 and 3.4.3.

2.4 Summary

In summary, this chapter has provided the necessary background information on the problem domain, wildlife management, and the solution technique, reinforcement learning, to understand the related work and methodology of this thesis. Section 2.1 gave a brief introduction to wildlife management, why it is important to ecosystems, and the appropriate actions for it. Section 2.2 presented the mathematical basis for simulating a biological ecosystem, by starting with a simple model and building upon it, making it more complex but also more realistic. Such an ecosystem simulation is necessary as it serves as an environment where an agent can test and improve its wildlife management skills. The agent is trained using the AI method of reinforcement learning. The final section of this chapter, section 2.3, serves as an introduction to reinforcement learning (RL) and deep reinforcement learning (DRL). It gives the reader the necessary fundamental knowledge to understand the related work in this field, as well as the DRL algorithms utilized in this thesis.

Chapter 3

Related Work

The following chapter is based on a similar chapter written in my specialization project (Singh, 2022). This chapter presents related work in the field of wildlife management and conservation, and reinforcement learning. To collect and select what work to be included in this chapter, a structured literature review was carried out. This process is described in section 3.1. Section 3.2 provides an introduction to some non-AI methods for wildlife management and conservation, which have been traditionally common. Section 3.3 presents RL methods used to facilitate wildlife management and conservation, which have become an increasingly popular research field. Section 3.4 covers some work done in similar problem spaces as this thesis, RL in grid environments. Finally, a summary of the work presented in this chapter as well as their value to this thesis is presented in section 3.5.

3.1 Structured Literature Review

This thesis was partly inspired by the *CAPTAIN* project (presented in section 3.4.4). The CAPTAIN project presents one way of utilizing RL for wildlife conservation. To gain some background knowledge of wildlife management and conservation, the relevant parts of Fryxell (2014) were read.

The literature search was done to get a better understanding of the traditional (non-AI) and AI approaches used in wildlife management. To do this, the search engines Google and Google Scholar were utilized. Initial searches and papers,

even though not necessarily relevant to this thesis, gave a better understanding of the vocabulary used in the field. This led to a collection of more specific keywords. For the literature search on traditional solutions for wildlife management, these keywords were used: *wildlife conservation*, *wildlife management*, *predator-prey*, and *spatio-temporal*. For AI approaches, these keywords were used: *Artificial Intelligence*, *reinforcement Learning*, *wildlife conservation*, and *spatio-temporal*. This search led to some papers on the intersection of RL and wildlife management, however, to get a deeper understanding of RL in grid environments where the state space consists of smaller cells, another search was done with the keywords: *Reinforcement Learning*, *grid* and *cell selection*.

When choosing what papers to include in this study, the following factors were looked at:

- Relevancy to the problem investigated in this thesis
- Publishing date
- Whether it was published in a journal
- Number of citations

Relevance to the thesis problem was naturally an important criterion. It was preferred that more recent research be included, no older than 20 years old. It was also advantageous if the paper had been published in a reputable journal, such as *Nature*. A higher number of citations was also preferred.

3.2 Traditional Approaches in Wildlife Management

3.2.1 Stochastic Dynamic Programming

A significant part of spatial conservation planning, a form of wildlife management, is to decide upon the geographical areas that are to be protected and conserved. Often times there is a restriction on the amount of area that can be protected as a consequence of limited resources. This means that there has to be a prioritization of what areas to protect. Wilson et al. (2006) prioritized the allocation of conservation resources to five regions in Southeast Asia when taking into account the different levels of biodiversity, threat, ongoing habitat destruction, and cost. The optimal solution to this problem was found using stochastic

dynamic programming (SDP). However, SDP was shown to be computationally intractable for more than a few regions, due to the "curse of dimensionality". Wilson et al. (2006) also solved the problem using two different heuristics which only looked one time-step ahead, making them computationally cheaper, and achieved results close to the optimal one in certain cases. This work shows that even ecosystems with simple dynamics can get computationally intractable with traditional methods (non-AI algorithms).

3.2.2 Spatio-Temporal Animal Reduction (STAR)

McMahon et al. (2010) created an interactive Microsoft Excel spreadsheet model to facilitate wildlife managers and researchers planning the control and reduction of an invasive species, named the *Spatio-Temporal Animal Reduction* (STAR) model. The STAR model aims to be a user-friendly and intuitive framework, which could also be used by non-experts. The model considers both the biological and financial aspects, as well as the spatial and temporal ones. Figure 3.1 shows a conceptual model of the STAR framework.

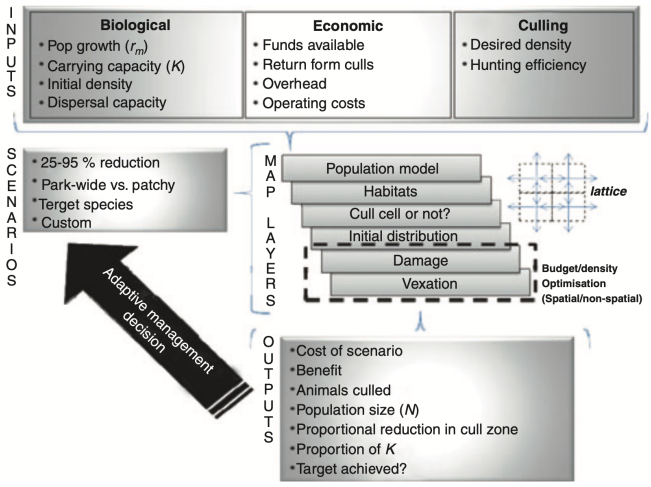


Figure 3.1: Conceptual model of the STAR framework. The user inputs biological, economic, and culling parameters. It is also possible for the user to specify spatial limitations such as whether or not a culling should occur in a cell. The framework will optimize the goal set by the user, and output whether or not it is possible to achieve the goal as well as the cost, number of animals culled, etc. (McMahon et al., 2010).

It is assumed in the model that the reduction of invasive species is done through *culling*. Culling is a process of reducing a wildlife population by killing some individuals. The biological input consists of population growth, carrying capacity, initial density, and dispersal capacity, while the economic input consists of the number of funds available, return from culls, overhead expenses, and operating costs. It is also possible to specify desired culling density and the efficiency of hunting. The STAR model has several *control layers*, which lets the user specify the geographical areas (represented as cells in STAR) that shouldn't be culled or that have high vexation values.

The user can set different management goals, that STAR will optimize. The user can get the maximum culling rate given a budget and a set of specified cells, or if a target density for an area is specified (referred to as the control area) if a budget and culling rate is sufficient to reach that density. There is also an option to minimize cost-benefit on a set of cells given a budget. These optimization problems are solved by iterating through the possible solutions, evaluating them, and choosing the one that performs best according to the relevant metric, such as cost-benefit. This brute-force approach works in the STAR model, as the solution space is discrete since the optimization algorithm only considers whole percentages when iterating through solutions. However, if the solution space was continuous, or the discrete solution space was larger, this approach would become computationally infeasible.

McMahon et al. (2010) applied the STAR framework on different invasive species in Kakadu National Park, Australia. Figure 3.2 shows an example of the framework being applied to cull pigs in a control area in the national park.

The STAR framework can easily be applied to other geographical areas with different species. For example, Wiggins et al. (2014) utilized the STAR framework on the problem of controlling the Tasmanian pademelons on agricultural land in Tasmania. The authors used it to compare the cost and benefits of different intensity levels of reduction. The relatively simple and intuitive framework makes it easy for non-experts to use it, however, it also means that some simplifications have been made. For example, the biological simulation of the system only accounts for one species and does not consider how culling animals of one species might affect the other species in the ecosystem.

Even though the methods used in this framework are not relevant to this thesis, it shows how wildlife management problems can be modeled. For example, it gives a better understanding of what factors to include when designing a reward function, state space, and action space in an RL environment. It also highlights the limitations of brute-force solutions.

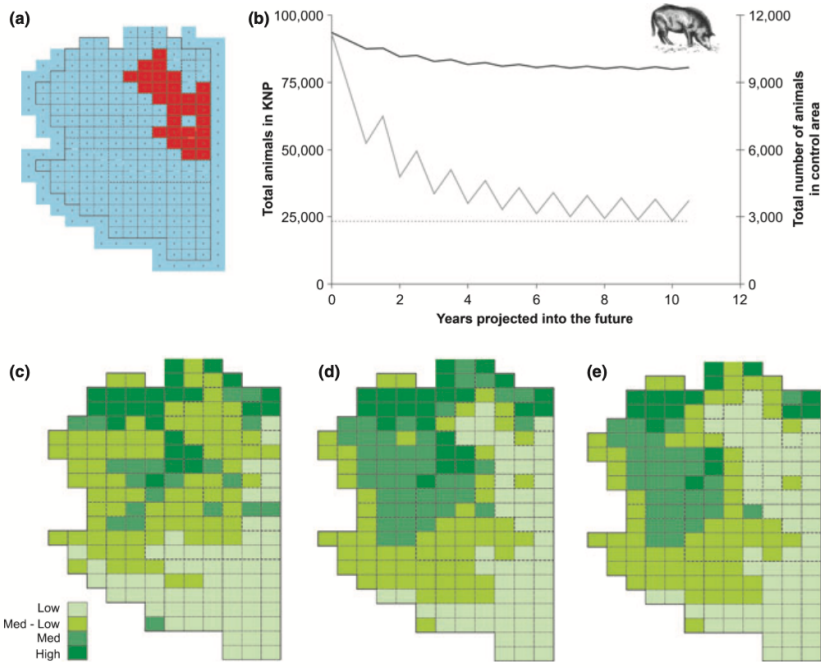


Figure 3.2: Example of STAR being applied to cull pigs in Kakadu National Park, Australia (McMahon et al., 2010). (a) Cellular map in representing the national park, control areas are red. (b) Projection of the pig population in the national park (dark gray) and the control area (light gray). (c) Distribution of pigs in the national park before culling. The higher the density of pigs, the darker the shade of green. (d) Distribution of pigs after culling in the control areas. (e) Distribution of pigs after culling in the control areas and adjacent cells.

3.2.3 Population Viability Analysis (PVA)

Fryxell et al. (2020) looked at how habitat fragmentation and loss in Ontario, Canada due to commercial logging and other human activities, affected the woodland caribou population growth. The research was conducted through a *population viability analysis* (PVA). PVA is a method that uses data about species in an ecosystem, such as annual survival rate and birth rate, with mathematical models to estimate their probability of survival over a longer time period. The study looked at both individual and spatially explicit PVA models. The study suggested that woodland caribou adult survival rates in logged landscapes were

declining as a consequence of wolf predation. This shows how changes in the spatial environment, such as habitat loss, can alter the dynamics of an ecosystem, and thus potentially lead to a steady decline of a species in that environment. To combat the population decline, Fryxell et al. (2020) suggested reducing the wolf density.

PVA is a useful tool when analyzing wildlife populations and estimating their future development, but it does not give any information on how active management decisions might affect the ecosystem. A software framework for this is presented by Nagy-Reis et al. (2020) and is called *WildLift*. WildLift estimates the performance of different management strategies relative to some given performance metric. Similarly to Fryxell et al. (2020) this study looks at an ecosystem consisting of caribou, moose, and wolf. The wolf preys on both moose and caribou, but the main aim of wildlife management is to protect the caribou population from declining. The software allows for different types of management actions, which are shown in figure 3.3. There are two types of habitat management actions: *Linear feature restoration* (LFR) and *Linear feature deactivation* (LFD). LFR aims at restoring the habitat of the ecosystem to its natural state through means such as tree planting. This can take several years. LFD attempts to slow down wolf movement speed, which can be done by blocking through fencing or tree-felling. Demographic augmentation is another type of management action and can be done in three ways: *Maternal Penning* (MP), *Conservation Breeding* (CB), and *Predator Exclosure* (PE). MP protects both adult female caribou and their calves from predation for a temporary period, so that their probability of survival is higher. CB is breeding caribou in captivity, and then releasing them into the wild, with the aim of increasing their abundance. PE are enclosed areas where predators and other competing species are removed so that the prey can breed and increase in abundance. A third type of management action is more short-sighted predator-prey management: Wolf Reduction (WR) and Moose Reduction (MR). WR intends to reduce the mortality rate of caribou by decreasing the population of the primary predator. MR aims to reduce the wolf population to reduce caribou mortality, by reducing the abundance of moose, which it preys upon.

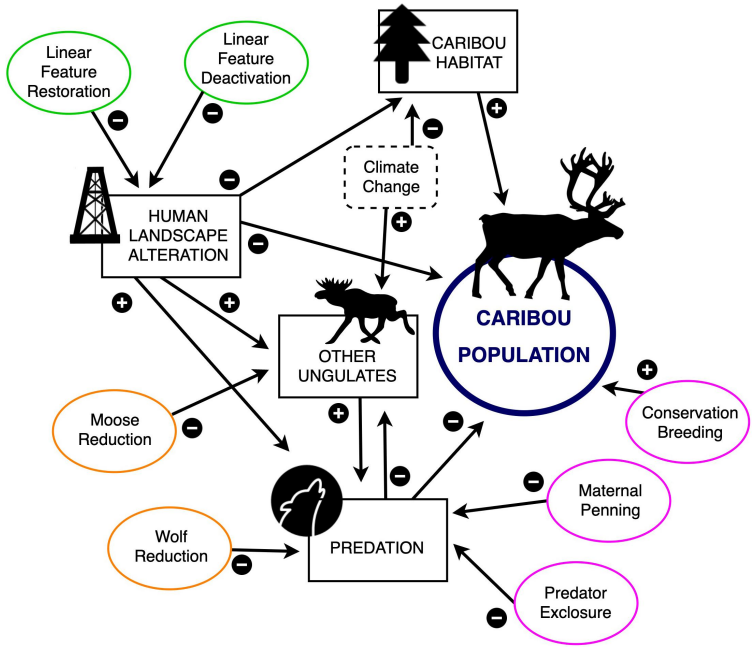


Figure 3.3: Diagram of the different management actions possible in WildLift, and how they affect the ecosystem. There are three types of management actions: Habitat management (green outline), Demographic augmentation (orange outline), and Predator-prey management (pink outline) (Nagy-Reis et al., 2020).

There are different ways to measure performance, depending on your aim. The performance measures used in this study were population growth rate, number of new individuals introduced, and also cost. The study evaluated the performance on these metrics of the different management actions by using PVA. Of the different management actions, only WR lead to an increase in the caribou population, suggesting that predator-prey management might be effective in stabilizing ecosystems short term.

In summary, PVA can be a valuable tool for wildlife managers, providing them with an estimate of the consequences of different actions on the population of a species. However, wildlife managers usually have multiple performance measures to keep in mind, and the number of actions can become large when accounting for the degree an action is to be applied, and where and when it is to be applied. This makes the already challenging task of wildlife management even more complex. This opens up the door for reinforcement learning.

3.3 Reinforcement Learning in Wildlife Management

3.3.1 Deep Reinforcement Learning for Conservation Decisions

Lapeyrolerie et al. (2021) explored the use of DRL in the domain of wildlife conservation. The authors highlighted some benefits of using RL in the domain of conservation compared to other ML methods. RL, in contrast to supervised and unsupervised learning, does not require large amounts of data to be collected. This is because the environment the RL agent interacts with serves as the source of data. Another advantage is that biologists and conservationists are familiar with creating and utilizing ecological simulations, which can serve as a fitting basis for RL environments.

To demonstrate the potential of DRL, the authors applied it to several conservation decision problems. Two of the problems were sustainable harvesting of fish and ecosystem management for avoiding tipping points. The DRL agent trained on these problems achieved impressive results. It is worth noting, as the authors also pointed out, that these problems were relatively simple. However, even though these problems were simple, they display the potential of DRL on conservation problems.

To create the RL environment in which the conservation problems took place, the authors used the OpenAI gym framework. Gymnasium, a framework based on OpenAI gym, was used to create the wildlife simulation in this thesis. A more detailed description is provided in chapter 4.

3.4 Reinforcement Learning in Grid Environments

Reinforcement learning in grid environments has generally revolved around training an agent to navigate through the grid, however, there have been a few applications of RL where the agent has been optimized to select cells. This subsection presents the DRL algorithms tested in this thesis, as well as some applications of them in cell-selection tasks.

3.4.1 Deep Q-Network

Deep Q-Network (DQN) is an algorithm to train a neural network based on the Q-learning method (described in section 2.3.3). The algorithm was first described by Mnih et al. (2013), who successfully applied it to play several Atari 2600 games. They represented the action-value function (Q-function) with a neural network and its parameters, θ. This neural network is referred to as the *Q-network*. The Q-network can be trained by minimizing the following loss function, L, after every iteration, i:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s,a;\theta_i))^2] \tag{3.1}$$

where $\mathbb{E}_{s,a\sim\rho(\cdot)}$ refers to the expected value when sampling state s and action a from the probability distribution $\rho(\cdot)$ based on the behavior policy. The term y_i is the TD-target: $y_i = \mathbb{E}_{s'\sim\varepsilon}[r + \gamma max_{a'} Q(s', a'; \theta_{i-1})|s, a]$. ε represents the environment the agent interacts with. The Q-value of the successive state-action pair in y_i is calculated using the parameters from the previous iteration, θ_{i-1}. The gradient of the loss function with regards to the Q-network parameters $\nabla_{\theta_i} L_i(\theta_i)$ could then be used to update the Q-network parameters. The equation for the gradient is given below:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}[(r + \gamma max_{a'} Q(s', a'; \theta_{i-1}) - Q(s,a;\theta_i))\ \nabla_{\theta_i} Q(s,a;\theta_i)] \tag{3.2}$$

∇_{θ_i} represents the gradient with regards to the Q-network parameters θ at time step i. The gradient of the loss function is equal to the expected value of the TD error multiplied by the gradient of the Q-value function (the Q-network). The expectations in this equation can be removed. This is possible as one could think of the single sample generated in a time step as drawn from the distributions ρ and ε, thereby converging towards the true expectations after many samples.

If the algorithm had trained the Q-Network on every transition sample generated each time step, it would have been *online* learning. However, the algorithm employs an *offline* learning method, where the Q-Network is trained on batches of randomly chosen samples that are collected in a replay memory. This method is also known as *experience replay*, and builds on ideas from the *Neural Fitted Q-Iteration* method presented by Riedmiller (2005). Offline learning has several advantages when training a neural network, as training with batches can lead to more stable learning and quicker convergence.

Mnih et al. (2015) built further upon their previous work on the algorithm. Similarly to the initial algorithm, the updated implementation of DQN uses experience replay: training on random batches of previous experiences. At each time step t the experience, defined as $e_t = (s_t, a_t, r_t, s_{t+1})$, is added to a dataset $D_t = e_1, ..., e_t$, which stored in a replay memory of finite size. The Q-network is trained using minibatches of experiences that are randomly sampled from this replay memory. This is an advantage as consecutive experiences often have a strong correlation, which could negatively affect the learning of the Q-network. Randomizing the experiences in a minibatch breaks this correlation, leading to less variance in updates to the Q-network.

One difference between the initial DQN algorithm and the updated one is the addition of a target network. Instead of generating Q-values used in the TD-target from the previous iteration Q-network parameters θ_{i-1}, the target network with parameters θ^- is used. Q-values generated by the target network are represented by the symbol $\hat{Q}$. The TD-target equation with the described changes becomes $y_i = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} \hat{Q}(s', a'; \theta^-)|s, a]$. This leads to the following gradient being the basis for stochastic gradient descent:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \varepsilon}[(r + \gamma max_{a'} \hat{Q}(s', a'; \theta^-) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)] \tag{3.3}$$

This equation is a variation of equation 3.2 but with the addition of a separate target network. The target network itself clones the Q-network periodically, meaning that after a set amount of time steps C, the target network is updated so that $\hat{Q} = Q$.

Since an update to the Q-network that increases a Q-value for one state-action pair $Q(a, s_t)$ would likely increase the Q-values of successive state-action pairs $Q(a, s_{t+1})$, it would increase the TD-target y_i. This could lead to the policy diverging. A time delay between updating the target network makes the correlation between the Q-values and the TD-target weaker, thereby making divergence less likely and the policy more stable. An overview of how the different components in DQN are used to train the algorithm is shown in figure 3.4.

Wang et al. (2018) applied Q-learning on the task of sparse mobile crowdsensing. The RL agent was trained to select one cell in the grid to collect data from every cycle (time step) to ensure a certain quality of the inferred data, with the aim of selecting as few cells as possible. A cell could either be selected for data collection or not, meaning that it had two possible states. The size of the state space would therefore become 2 to the power of the number of grid cells.

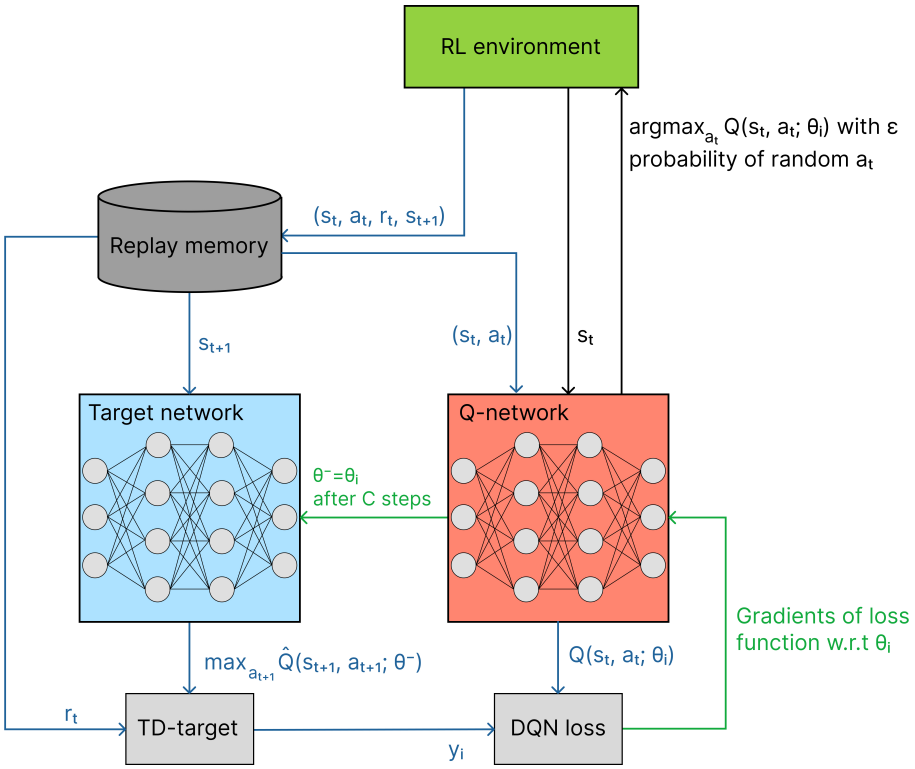


Figure 3.4: Overview of DQN architecture with a target network. The black arrows show how the Q-network chooses an action based on the current state s_t. Once the action has been performed in the environment, the experience is stored in the replay memory. From the replay memory, minibatches of random experiences are trained on. To train the Q-network, the loss needs to be computed. The blue arrows represent the flow of information leading to the calculation of DQN loss. Gradients based on this loss are then used to update the Q-network. For every C number of steps, the target network copies the Q-network.

The authors used Q-learning to optimize the agent on the task and tested two approaches to learning the Q-function, one being *table-based* and the other based on a *Deep Recurrent Q-Network* (DRQN). Tabular Q-learning aims to learn the Q-function as a table. This approach has been used in traditional RL algorithms, however, the state space grows exponentially with regard to the grid size, meaning that the table storing the Q-values would become intractable for large grid problems. The second approach, DRQN, deals with this problem, also referred to

as the "curse of dimensionality", by utilizing a deep neural network to learn the Q-function. Instead of dense layers, LSTM (Long-Short Term Memory) layers were used between input and output in the network, to capture the temporal aspect of the problem. DRQN achieved better results than other, widely used cell selection algorithms, displaying the potential of Q-Networks to learn good value functions in cell-selection tasks.

3.4.2 Actor-Critic

Following the success of DQN on Atari 2600 games, Mnih et al. (2016) applied *asynchronous* reinforcement learning methods to the same domain and achieved significantly better results. The idea behind asynchronous methods is to asynchronously run multiple *workers* in parallel, often each on their own CPU thread, and each interacting with their own instance of the RL environment. Each worker can then make changes to the globally shared parameters according to their experiences. This makes the experience replay used in DQN, no longer necessary. Even though experience replay has its advantages when training neural networks, it has the memory cost of having to store transitions. It also cannot, by definition, be used in on-policy algorithms. Asynchronous methods avoid these drawbacks and also allow different workers to explore different parts of the environment at the same time. This is beneficial as it makes the data more diverse, instead of only being determined by which parts of the environment a single agent has explored. Diverse data lead to more robust training. Since asynchronous methods run workers in parallel, they also have the practical advantage of faster training. Four different asynchronous methods were tested: one-step SARSA, one-step Q-learning, n-step Q-learning, and advantage actor-critic. Out of these, the *Asynchronous Advantage Actor Critic* (A3C), the only policy-gradient method, performed best.

A3C is similar to the actor-critic method described in section 2.3.5, but has a couple of fundamental differences. The first difference is that there are multiple workers running simultaneously and calculating different gradients to update the parameters. The gradients used to update policy and value function parameters are accumulated from all these different workers, rather than experiences from one single agent. The second difference is that A3C is n-step TD, while the actor-critic method described in section 2.3.5 and the DQN algorithm is one-step TD.

A3C updates parameters using the gradient estimator from equation 2.17, where the advantage function is approximated with the TD-error, similarly to the actor-critic method described in section 2.3.5. The advantage function is therefore given

by the following equation:

$$A(s_t, a_t; \theta, \theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v) \tag{3.4}$$

The policy parameters are represented by θ, and the value function parameters are represented by θ_v. k is the number of states from the starting state to the final state. This is upper bound by t_{max}, which is equal to the step size n in the n-step TD. If a terminal state in the environment is encountered before n states after the starting state has been visited, k will be lower. The sum and the second term in the equation make up the n-step return (same as equation 2.11). With the subtraction of the estimated value of the start state, the equation becomes the n-step TD error.

Based on the advantage function and the actor-critic policy gradient estimator from equation 2.17, the equation for accumulating policy parameters gradients and value function gradients in A3C is given as:

$$d\theta = d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v)) \tag{3.5}$$

$$d\theta_v = d\theta_v + \partial(R - V(s_i; \theta'_v))^2 / \partial\theta'_v \tag{3.6}$$

where R is the TD target:

$$R = r_i + \gamma r_{i+1} + ... + \gamma^{k-1} r_{i+k-1} + \gamma^k V(s_{i+k}; \theta'_v) \tag{3.7}$$

θ are the global policy parameters while θ' are the policy parameters specific to each worker. Likewise, θ_v are the global value function parameters and θ'_v are the worker-specific value function parameters. The gradients are accumulated for each state s_i between the starting state and the final state. Since R represents the TD-target, the term $(R - V(s_i; \theta'_v))$ equals the TD-error, which is the advantage function used in A3C (equation 3.4). Similarly to DQN, the loss for the critic, or value function, is the mean squared TD error.

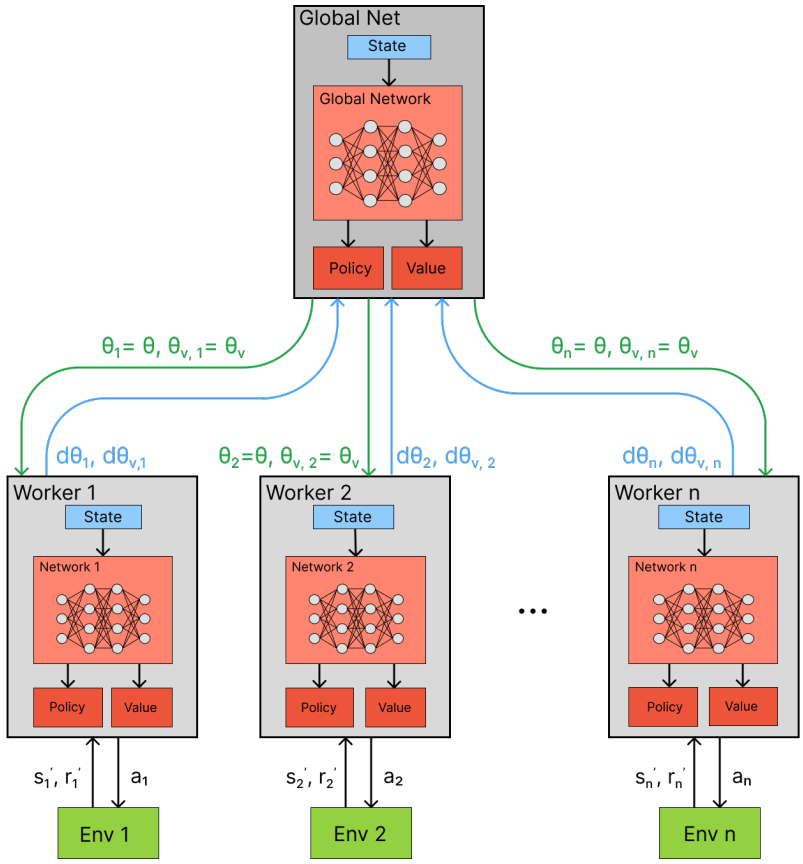


Figure 3.5: An overview of the A3C algorithm. Multiple workers interact with their own instance of the environment in parallel, as shown by the black arrows. After a fixed number of steps each worker i accumulates gradients, $d\theta_i$ and $d\theta_{v,i}$, with regards to its policy θ_i and value function $\theta_{v,i}$. These are asynchronously sent to the global network to update it, as shown by the blue arrows. Once the global network has been updated, the worker copies its parameters, indicated by the green arrows.

With their own accumulated gradients $d\theta$ and $d\theta_v$, workers update the global parameters θ and θ_v asynchronously. Even though the policy and value functions have been described as having unique parameters, it is usual for both to be calculated by the same neural network. All layers are shared, except for the output layers. Figure 3.5 shows an overview of A3C.

Wu et al. (2017) presented the *Advantage Actor Critic* (A2C) algorithm, a synchronous variant of the A3C algorithm, which the researchers found to perform better than A3C. Since it is synchronous, A2C waits for all workers to collect their portion of experiences before applying a global update. This has the advantage of efficient GPU use since the gradients of all workers can be calculated at once, in large batches. Efficient GPU use could lead to faster training times.

The A2C architecture is similar to the architecture of A3C, which is shown in figure 3.5. However, since A2C is asynchronous, each worker will send its gradients to the global network at the same time, and thereafter also copy the updated global network at the same time. This means that all workers essentially have the same policy, which was not the case in A3C. Since all the workers have the same policy, it is possible to instead only have the global policy interact with multiple environments, each having different starting states. This is known as a *vectorized environment*. After interacting with all the environments for a fixed amount of steps, gradients of the network could be calculated based on the batch of experiences collected. The gradients would then be used to update the global network.

Altamimi et al. (2022) investigated the use of deep RL for large-scale wildfire mitigation. Forest wildfire was formulated as an MDP, where the state space consisted of a grid of cells representing the spatial structure of a forest. Each cell represented a forest stand. Fire could propagate through neighboring cells, and was affected by the wind direction. At each time step, the agent performed an action on one cell, and in the following time step, performed an action on the next cell. The agent could either harvest a cell by removing all timber or do nothing and let the timber grow. For each year the volume of timber in a cell grew. The reward grew with the timber volume and when the risk of wildfire decreased. An off-policy actor-critic model was tested on this task, with grids of different sizes, and was compared with other approaches. Figure 3.6 shows the performance of actor-critic and DQN on a 3×3 grid. The actor-critic algorithm manages to obtain the optimal reward, while DQN does not.

For larger grids, actor-critic was compared to genetic algorithms (GA). In both the 10×10 and 100×100 grid, the actor-critic managed to outperform GA. These problems were too complex to solve with the value iteration algorithm.

These results showcase the potential of actor-critic methods in grid environments, even outperforming DQN. It is worth noting that the actor-critic model implemented in this paper utilized experience replay, such as in DQN. An alternative actor-critic approach, such as A3C or A2C, might have yielded different results.

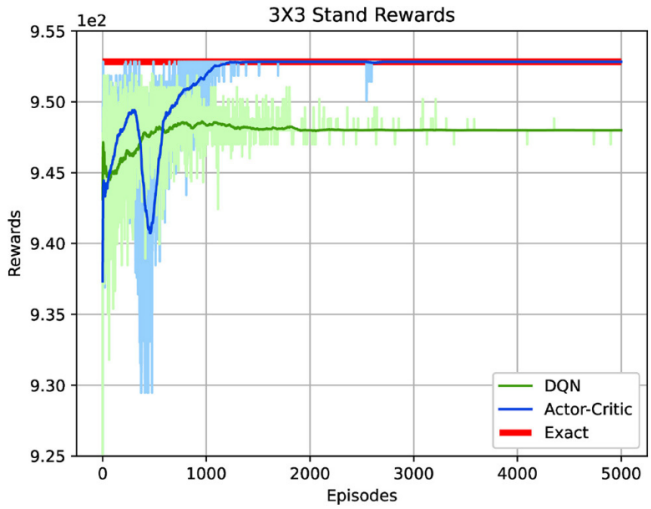


Figure 3.6: Performance of actor-critic and DQN on mitigation of wildfire in a 3×3 grid (Altamimi et al., 2022). The x-axis shows the number of episodes, and the y-axis shows the rewards obtained. The optimal solution is shown by the red line, and found by solving the Bellman equation with a value iteration algorithm.

3.4.3 PPO

Schulman et al. (2017a) created the *Proximal Policy Optimization* (PPO) algorithm, which is based on the *Trust Region Policy Optimization* (TRPO) algorithm (Schulman et al., 2017b). The TRPO algorithm is a trust region method, with the idea behind it being that the new policy of the agent should be close to its old policy. In other words, there should not be any significant updates when optimizing the policy of the agent.

While traditional policy gradient methods, such as A3C and A2C, use the policy gradient estimator from equation 2.17 to optimize performance, TRPO tries to maximize the following "surrogate" objective function:

$$L^{CPI}(\theta) = \max_{\theta} \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t] \tag{3.8}$$

where $L^{CPI}(\theta)$ is the objective with regards to the policy parameters θ. CPI stands for "conservative policy iteration". π_θ represents the policy we want to

change to maximize the objective, and $\pi_{\theta_{old}}$ represents the old policy. $\hat{\mathbb{E}}_t$ represents the empirical average over a finite batch of experiences. A_t represents the advantage function at time t. This objective is subject to the following constraint:

$$\hat{\mathbb{E}}_t[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta \tag{3.9}$$

where $KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]$ is the *Kullback–Leibler* (KL) divergence, which is used as a measurement of the distance between the two probability distributions $\pi_{\theta_{old}}$ and π_θ. To satisfy the constraint, the distance needs to be less or equal to the hyperparameter δ. TRPO has been shown to perform better or equal to other RL algorithms such as DQN, while also demonstrating stable policy improvements.

PPO was created with the motivation to reduce the complexity and overhead of TRPO while utilizing its stability and performance. The clipped "surrogate" objective function proposed in PPO is given below:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{3.10}$$

where ϵ is a hyperparameter (typically $\epsilon = 0.2$) and $r_t(\theta)$ is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

which means that $r_t(\theta)\hat{A}_t$ is equal to L^{CPI} (equation 3.8). This objective function takes the minimum of the unclipped (first term) and clipped (second term) objective L^{CPI}.

To get an intuitive understanding of this objective function, one could think of $r_t(\theta)$ as the change in probability of taking action a_t when going from policy $\pi_{\theta_{old}}$ to π_θ. If $r_t(\theta) > 1$ then action a_t is more likely to be chosen under the new policy π_θ, compared to the old policy $\pi_{\theta_{old}}$. Likewise, if $r_t(\theta) < 1$ then action a_t becomes less likely to be chosen under the new policy, and if $r_t(\theta) = 1$, there is no change in the probability of choosing a_t. As with TRPO, it is desired that updates to the policy are non-significant. That is why the second term in the minimization operation is introduced. It clips $r_t(\theta)$ to be between $1-\epsilon$ and $1+\epsilon$. To get a better understanding of how this affects the objective function, one could look at the two scenarios where the advantage A_t is non-zero.

If the advantage is positive it means that the action taken a_t is better than what is expected of being in state s_t. It would therefore be preferable to have a higher

probability of choosing this action in the next policy. If the new policy is less likely, or only slightly more likely to choose this action compared to the old policy, the gradient of the objective function is positive, as seen in figure 3.7. However, if $r > 1 + \epsilon$ the new policy is significantly more likely to choose this action than the old policy, and making the succeeding policy even more likely to choose this action could be an overly optimistic approach. Therefore, no updates are made if r becomes larger than $1 + \epsilon$. This is done by clipping r when it becomes larger than $1 + \epsilon$, resulting in the gradient of L^{CLIP} becoming 0.

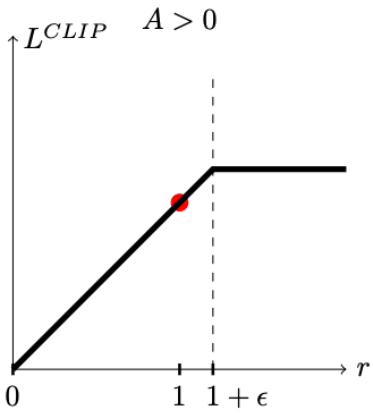


Figure 3.7: The value of the clipped "surrogate" objective function L^{CLIP} with respect to $r_t(\theta)$ when the advantage A_t is positive (Schulman et al., 2017a). The x-axis represents the value of $r_t(\theta)$, and the y-axis represents the value of L^{CLIP}. r gets clipped when it becomes greater than $1 + \epsilon$.

If the advantage is negative, the action taken a_t is worse than what is expected of being in state s_t, and it is, therefore, desirable to decrease the probability of it being selected in the new policy. However, if r is less than $1 - \epsilon$ it means that the new policy is significantly less likely to choose this action compared to the old policy, and further decreasing its probability of being chosen could lead to a too significant (and too pessimistic) of an update. Therefore, the objective function is clipped when $r < 1 - \epsilon$, as visualized in figure 3.8. If the new policy is slightly less likely to choose a_t, further decreasing the probability of choosing it in the succeeding policy would not lead to a significant update overall. Therefore, the gradient of the objective function is negative when $1 - \epsilon < r < 1$. If the new policy is significantly more likely to choose this action than the old policy, it is desirable to "roll back" this update (between the old policy and the new one). This is done by not capping the L^{CPI} objective function when $r > 1$.

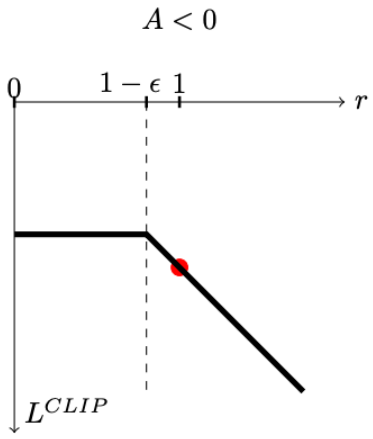


Figure 3.8: The value of the clipped "surrogate" objective function L^{CLIP} with respect to $r_t(\theta)$ when the advantage A_t is negative (Schulman et al., 2017a). The x-axis represents the value of $r_t(\theta)$, and the y-axis represents the value of L^{CLIP}. r gets clipped if it is less than $1 - \epsilon$.

Given the objective function L^{CLIP}, a loss function for PPO can be constructed. Assuming that PPO utilizes an architecture similar to A2C, where the policy function and value function share layers in a neural network, the loss needs to include a term for the value function error. The value function error L_t^{VF} is defined as:

$$L_t^{VF} = (V_\theta(s_t) - V_t^{targ})^2$$

where $V_\theta(s_t)$ is the estimated value of state s_t according to value function, and V_t^{targ} is the TD-target similar to the one in A2C (equation 3.7).

The PPO loss function also includes an *entropy bonus* term that improves the exploration of actions by encouraging the action probability distribution of the policy to be more random. In the case of RL problems with a discrete action space, as is the case in this thesis, the entropy bonus term $S[\pi_\theta](s_t)$ is often defined as the *Shannon entropy* of the action probability distribution:

$$S[\pi_\theta](s_t) = H(\pi_\theta(\cdot|s_t)) = -\sum_{a \in A} \pi_\theta(a|s_t) \log \pi_\theta(a|s_t)$$

where π_θ is the action distribution and essentially the policy, a represents an

action from the set of all possible actions A, and s_t represents the state of the environment at time t.

The final PPO loss function becomes:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{3.11}$$

where c_1 and c_2 are hyperparameters deciding the weight of the value function error and the entropy, respectively.

Notice that PPO is a policy optimization technique, similar to the policy gradient technique used in A2C and A3C (section 3.4.2). It is therefore possible, to use PPO as the implementation of the actor in actor-critic methods.

Schulman et al. (2017a) implemented it in an architecture similar to A2C when evaluating its performance. PPO outperformed A2C, TRPO, and other algorithms on several advanced continuous control environments, while also displaying stability and reliability. This has made PPO very popular. It is widely used, and considered as a state-of-the-art RL algorithm.

3.4.4 Evolutionary Strategies

Silvestro et al. (2022) tackled the problem of spatial conservation prioritization with the aim of protecting biodiversity, using RL. They created a framework named *CAPTAIN* (Conservation Area Prioritization Through Artificial INtelligence). The framework environment is a 2D grid representing a geographical area. The environment simulates biodiversity through aspects such as mortality, replacement, and dispersal. The grid is built up of smaller cells that contain information about that specific subarea. The environment contains information about multiple variables, such as species richness, population density, anthropogenic disturbance, climate, and more. The framework optimizes a conservation policy (also called protection policy) according to some biodiversity target decided by the user, while also accounting for financial restrictions such as a limited budget. The conservation policy, which is represented in the form of weights in a neural network, tells the agent what areas to protect. This is done by protecting *protection units*. A protection unit is an area consisting of multiple cells, which if protected, is shielded from exterior influence such as anthropogenic disturbance.

Conservation policy optimization happens with respect to a reward, which can be defined in various ways depending on the criterion for optimality. If optimal behavior is defined to be the restriction of species extinction, the reward could be determined by the number of species that did not go extinct at the end of a simulation. An overview of the CAPTAIN framework is shown in figure 3.9.

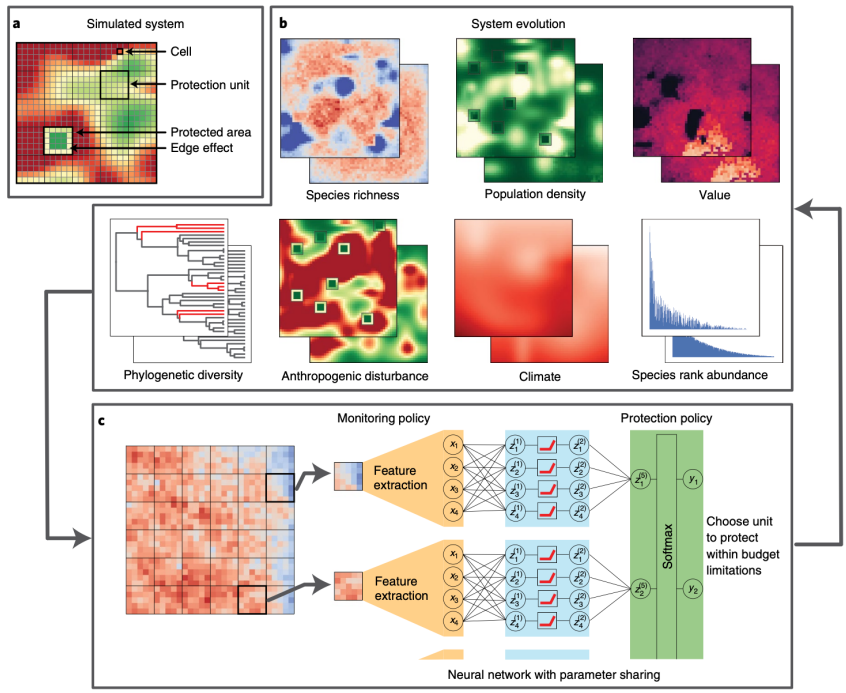


Figure 3.9: Overview of the CAPTAIN Framework (Silvestro et al., 2022). a) The simulated system consists of cells. A protection unit consists of multiple cells. b) Different variables of the system. c) Features extracted from each protection unit, fed through the neural network and given a probability. The neural network represents the conservation policy of the framework.

The parameters of the neural network are optimized to maximize the expected reward received from the protection actions. This is accomplished through a variation of *Parallelized Evolutionary Strategies*. Evolutionary strategies (ES) are black-box optimization algorithms inspired by natural evolution. Salimans et al. (2017) explored the use of ES in RL problems and gave an introduction to how ES functions. The algorithms iterate over two phases:

1. Randomly perturb the parameters of a policy, multiple times, and evaluate the different policy's performance by running an episode in the environment.
2. Combine the results of each policy and its results from the episode, calculate a stochastic gradient estimate, and update the parameters.

One of the advantages of ES is that they are highly parallelizable. Since the evaluation of policies with different parameters is independent of each other, multiple workers can run in parallel. In addition to being parallelizable, the ES algorithm also has the advantage of not needing to perform backpropagation. This means that there is no need to calculate gradients, which can be computationally expensive. Other issues are also avoided, such as exploding/vanishing gradients.

While ES has many advantages over traditional RL algorithms, they seem to be less effective at RL problems where actions and rewards have a high correlation. When the rewards are sparse, and there is little correlation between individual actions and the reward, ES shows promising results compared to other RL algorithms.

The CAPTAIN framework showed promising results, being able to compete with the state-of-the-art algorithms on the spatial conservation prioritization problem. It also shows how a spatio-temporal wildlife simulation can be modeled as an RL environment.

3.5 Summary

The related work presented in this chapter has been found and selected through a structured literature review, which is described in section 3.1. Section 3.2 presented some non-AI approaches to wildlife management. These approaches are relatively simple and are restricted to smaller problem spaces. However, they give insight into how wildlife management problems can be modeled, which in turn has been helpful when implementing the RL environment in this thesis. For example, some of the studies have used 2D grids with varying colors representing

population density. This serves as a simple, but effective way of visualizing the spatial aspect of the environment, and is used in this thesis.

Even though RL has not been extensively used in the field of wildlife management, it has some advantages. These are discussed in section 3.3. These advantages also apply to the wildlife management problem in this thesis.

More detailed use of RL is described in section 3.4, which presents different RL algorithms and discusses how some of them have been successfully applied to other grid-environment problems. Since limited work has been done in applying RL to wildlife management problems, some of the work discussed in this section contain other problem domains. However, the problem space remains similar to the one in this thesis, a 2D grid where the RL agent has to choose one or more cells to apply an action to. It is useful to see which RL algorithms have been applied to similar problem spaces and how they perform, as these would have high applicability to the problem tackled in this thesis.

Chapter 4

Methodology

This chapter presents the methodology for creating the system that was used to answer the research questions posed in this thesis. An overview of the system built is presented in section 4.1. The system consists of two major components: The spatio-temporal wildlife environment, which is described in section 4.2, and the RL agent. The RL agent is trained using different DRL algorithms, which are presented in section 4.3. Section 4.4 describes how the DRL algorithms were trained and tested on the wildlife environment to obtain the results in the subsequent chapter. Finally, section 4.5 summarizes this whole chapter.

4.1 System Overview

To investigate and answer the research questions posed in this thesis, a software system was created. The system was coded using the programming language **Python**, which is simple, flexible, and has a large number of libraries well suited for AI programming, such as **NumPy** and **SciPy**. The system can be divided into two major components: A spatio-temporal wildlife environment, which serves as the RL environment, and the RL agent interacting with it. The wildlife environment simulates the physical system of wildlife, with interactions between species over time and space. The RL agent is trained with a DRL algorithm to perform actions in this environment, with the goal of maintaining a diverse system. Different DRL algorithms can be used to train the agent. A high-level overview of the system is shown in figure 4.1. The system is naturally shaped after a general RL system, shown in figure 2.6.

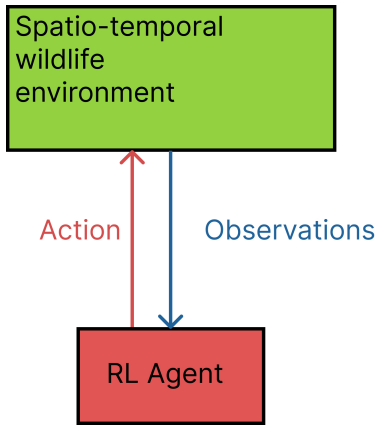


Figure 4.1: A high-level overview of the system built to answer the research questions posed in this thesis. The two major components of the system are the spatio-temporal wildlife environment and the RL agent. The RL agent is trained by a DRL algorithm to perform actions on the environment. It receives observations from the environment afterward.

The two major components of the system are described in the following sections.

4.2 Spatio-temporal Wildlife Environment

The spatio-temporal wildlife environment was created from the ground up, as this made it easier to have full control over the system behavior, whilst also making it easily modifiable. The environment consists of three smaller components: *BioEnvironment*, *Renderer*, and *BioGymWorld*. Each of the components handles different aspects of the RL environment and its interfaces.

An overview of the spatio-temporal wildlife environment and its smaller components is shown in figure 4.2.

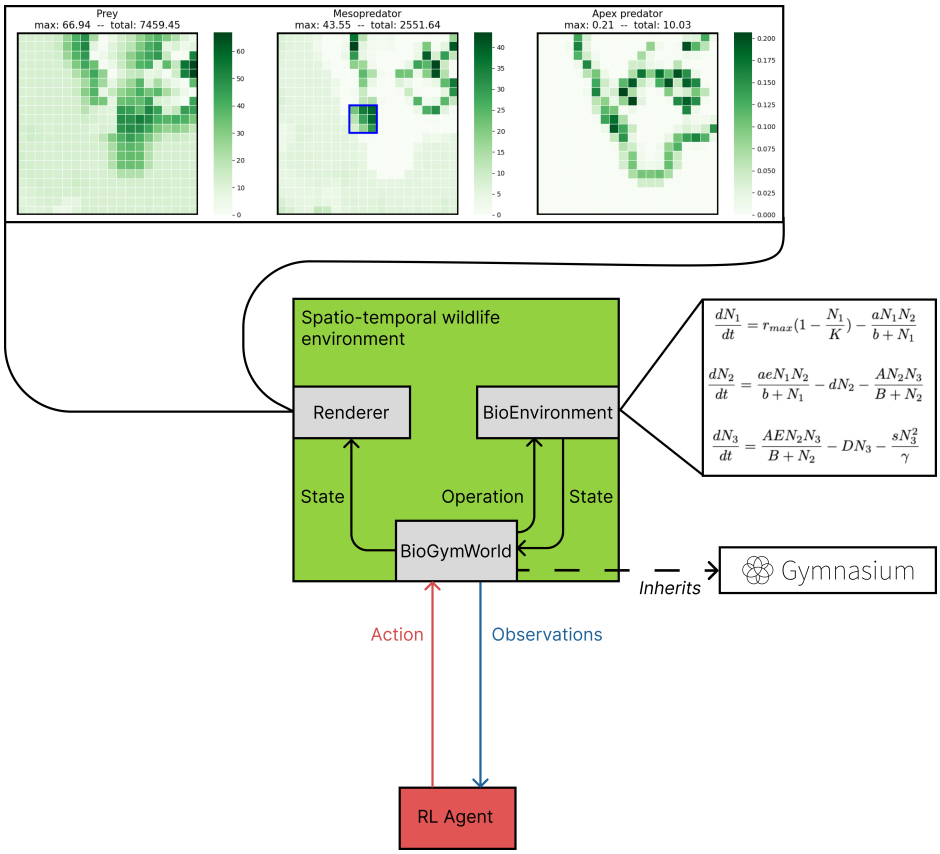


Figure 4.2: Detailed overview of the spatio-temporal wildlife environment component. The component consists of three smaller components: BioGymWorld, BioEnvironment, and Renderer. BioGymWorld, which inherits from Gymnasiums **gymnasium.Env** class, serves as an interface between the two other components and the RL agent. BioEnvironment hosts the logic for the wildlife simulation, while the Renderer renders a graphical representation of the simulation. Solid lines represent the flow of information. The dashed line represents inheritance. An "operation" includes the action decided by the RL agent, resetting the environment, and a time step in the simulation.

4.2.1 BioEnvironment

BioEnvironment hosts the biological ecosystem logic for the simulation. It holds the state of the system at every time step and the logic for state transitions. It models a tri-trophic system, as described in section 2.2.

Inspired by the cellular map representation used in the CAPTAIN project and the STAR framework (both discussed in chapter 3), the tri-trophic system is represented as three separate two-dimensional grids, one grid for every species. Since the three different grids represent the same geographical area, they can be thought of as different layers of the area, each displaying the spatial distribution of the population of a specific species. The grids consist of cells, which represent species populations in smaller areas within the grids. This representation of the system captures the spatial aspect of the environment, while also making it easier to get an overview of the entire system.

The BioEnvironment component stores and operates on the different species population grids as matrices. The matrices are graphically represented as population density grids by the Renderer component (more details in section 4.2.2). Both representations of the species populations are shown in figure 4.3.

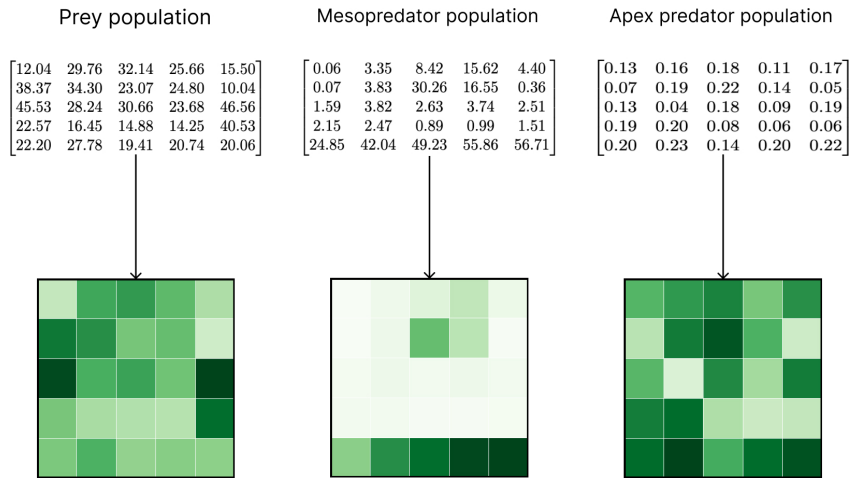


Figure 4.3: A 5×5 environment with the two types of species population representations: Matrices, as stored in the BioEnvironment component, and population distribution grids, as rendered by the Renderer component. Darker cells in the grids represent a higher population density.

Each time a BioEnvironment component is created or reset, the cell populations of the different species are randomly set within their respective *species range*. A species range is the range within which the population of a cell is randomly set. For example, if the species range for the prey species is 0 to 70, each cell in the prey population grid would be assigned a random number picked from the continuous uniform distribution between 0 and 70.

After the BioEnvironment has been initialized, actions can be applied to it and biological aspects can be simulated. This is done through the **step** function, which follows the general RL framework shown in figure 2.6 in that it takes in an action, simulates a time step in the wildlife environment, and ends up in a new state. The step function can be broken down into the following substeps executed in the given order:

1. Apply the action given as input
2. Simulate a time step in the tri-trophic system
3. Simulate species extinction
4. Simulate species dispersal

A more detailed description of each of these substeps is given in the following subsections.

Actions

There are multiple approaches to preserving biodiversity in an ecosystem, and some of these were presented in chapter 3. For example, in the CAPTAIN project, the action set consisted of protecting protection units, which shielded the species inside from external disturbance. This is a popular approach to wildlife conservation and is known as custodial management. The actions in BioEnvironment are manipulative rather than custodial. These two terms were introduced in section 2.1. Put simply, manipulative management means that the species' populations are directly affected by the actions. More specifically, the action set consists of directly adding or removing individuals of a specific species in a specified area.

Manipulative management makes the connection between the action and subsequent state clear for the RL agent. It has also shown to be an effective way to increase wildlife population, as discussed in section 3.2.3. Manipulative management might also be more suitable for an RL agent, as it might perform multiple

actions over time without restriction. When dealing with custodial management, however, an RL agent would normally be limited in the number of segments of an area it could protect.

The action set in BioEnvironment consists of placing *action units*. An action unit is a fixed-size square area that can be placed anywhere inside the grid for any species. The action unit can either decrease or increase the species population of cells it covers. If the action is to decrease the population, the population of every cell covered by the action unit is set to zero. If the action is to increase population, every cell covered by the action unit receives a single fixed increase of individuals of that specific species. In this case, every cell inside the action unit receives a population equal to the extinction threshold for that species times an *action multiplier*. Extinction thresholds are presented in the subsection regarding species extinction. The action multiplier is a hyperparameter of the simulation. The agent can only place one action unit at every time step and has the option of not placing any action unit at all. Placing an action unit negatively affects the rewards the agent gets, to resemble the real-life cost of removing or adding new individuals to an ecosystem. The reward function of spatio-temporal wildlife environment is discussed in further detail in section 4.2.3. Figure 4.4 shows a graphical representation of two action units being placed, one removing population and the other adding population.

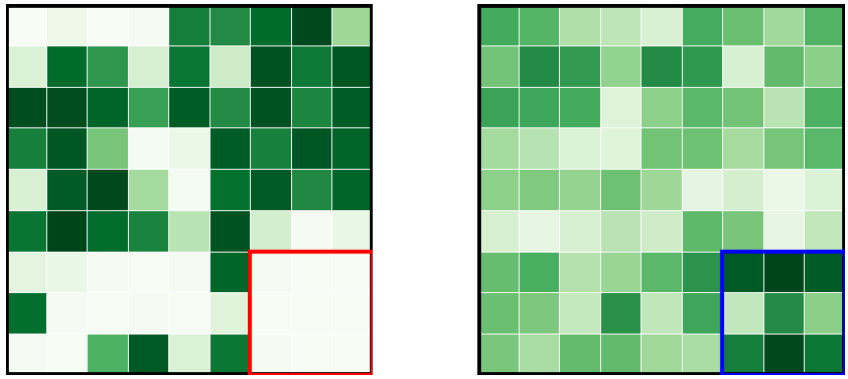


Figure 4.4: Graphical representation of a 3×3 action unit placed in the bottom-right of two different 9×9 environments. In the environment on the left, the action unit (red square) removes all individuals of that specific species from cells covered by the action unit. On the right, the action unit (blue square) adds a fixed number of individuals of a certain species to the cells it covers. Darker cells in the grids represent a higher population. The action units can be placed anywhere inside the grid, and can either remove or add individuals.

The number of places to place an action unit depends on the environment grid size and action unit size, and is given by the following equation:

$$num\_action\_unit\_places = (grid\_size - action\_unit\_size + 1)^2 \tag{4.1}$$

The action units placed can either increase or decrease the species population of any of the three species in the ecosystem. The RL agent additionally has the option of not placing an action unit. These possibilities result in the following equation for the number of possible actions:

$$num\_actions = (num\_action\_unit\_places \times 6) + 1 \tag{4.2}$$

where *num_action_unit_places* is given by equation 4.1.

Action units are inspired by the protection units in the CAPTAIN project (section 3.4.4). Protection units were, similar to action units, square areas of cells that the RL agent could perform an action upon. Such an action set was shown to be practical as it did not make the action space too large or too small.

Placing an action unit that removes individuals of a species is realistically possible and practical action as shown by the STAR framework (section 3.2.2). In the STAR framework, it was assumed that animals were removed through culling, but, in practice, it is also possible to remove them from the given ecosystem and to another one where they are more needed.

From a real-world perspective, placing an action unit that adds population could be thought of as *species reintroduction*: Releasing individuals of a species who have lived in captivity back into their ecosystem, also known as *ex situ*, or moving wildlife populations between areas, also known as *in situ*. It is worth mentioning that species reintroduction, especially ex situ, might negatively influence the animal's chances of surviving in the new area, as it has to adapt to it. As it is difficult to predict or model this negative influence, it is not considered in this thesis.

Research question 1 is posed with the intention of finding the action set that allows the RL agent to perform best. The action set has two modifiable parameters: the size of the action unit, and the action multiplier. The action set can be changed by altering the values these parameters inhibit. A select set of values for these parameters were tested in experiments, and are given in table 4.1.

Tri-trophic System

To simulate a time step in this tri-trophic system, the equations presented in section 2.2 are simulated for each cell in the grid. To simulate the ordinary differential equations in a cell, the population of each species in that cell is used. These are given by indexing that specific cell in their respective grids.

To reduce the complexity of the problem, the tri-trophic system is stabilized, thus not displaying chaos. The equations simulated for each cell are reiterated below for convenience:

$$\frac{dN_1}{dt} = r_{max}(1 - \frac{N_1}{K}) - \frac{aN_1N_2}{b + N_1} \tag{4.3}$$

$$\frac{dN_2}{dt} = \frac{aeN_1N_2}{b + N_1} - dN_2 - \frac{AN_2N_3}{B + N_2} \tag{4.4}$$

$$\frac{dN_3}{dt} = \frac{AEN_2N_3}{B + N_2} - DN_3 - \frac{sN_3^2}{\gamma} \tag{4.5}$$

For each time step in the tri-trophic system, two time steps of the tri-trophic equations are simulated. This was done to get a more dynamic simulation.

Species Extinction

To mimic extinction and avoid the atto-fox problem described at the end of section 2.2, there are *extinction thresholds* for each species. If the population of a species in a cell drops below its respective threshold, the population in that cell is set to zero. These extinction thresholds are set implicitly and are based on the carrying capacity for prey, and death rates for predators:

- Prey: 5% of K
- Mesopredator: d
- Apex predator: 2.5% of D

The parameters K, d, and D are modifiable parameters in the tri-trophic system. These thresholds were set through small-scale testing with the intention of having a lively ecosystem where species did not go extinct too often but had a small probability of it.

Species Dispersal

A significant aspect of the spatio-temporal wildlife environment is species dispersal. As time goes by, species travel across space to find more food. To simulate biological dispersal a fixed percentage, named *dispersal rate*, of a species population in a cell disperses to neighboring cells. An equal amount of individuals move to each neighbor. Neighbors are defined as adjacent cells that share an edge. The environment does not allow for "wrap-around", meaning that species are not able to move across grid edges to the opposite side of the grid. Dispersal is simulated for all species in all cells at every time step. The dispersal rate is a hyperparameter and can be different for each species. It does not change during a run of the simulation. Figure 4.5 illustrates how dispersal works for different cells.

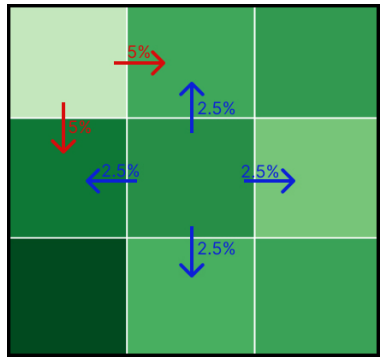


Figure 4.5: Graphical representation of dispersal out of 2 cells in a 3×3 environment. The dispersal rate is set to 10%, meaning that 10% of the population in each cell disperses out to neighboring cells. Since there is no "wrap-around", corner cells add 5% to both its neighbors, as shown by the red arrows. Cells that are not on the edge of the environment, move 2.5% of their population to each of their 4 neighbors, as shown by the blue arrows. Diagonal cells are not counted as neighbors.

4.2.2 Renderer

The Renderer component provides a graphical representation of the state of the wildlife simulation. This makes it easier to get an intuitive understanding of how the system evolves over time. It is also useful for debugging. The component shows a grid for each of the different species. The grids display the population

density in the different cells of the grids. There is also information about the highest population in a single cell as well as the total species population in the whole grid.

The species population density grids are displayed using the **heatmap**-function in **Seaborn**, which is a library based on the Python library **matplotlib**. Figure 4.6 shows the graphical representation of the BioEnvironment produced by the Renderer.

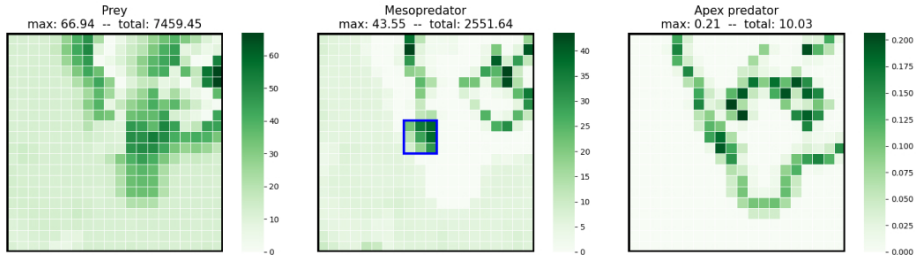


Figure 4.6: Graphical representation of a 20×20 environment. Each species is represented by its own heatmap and corresponding colorbar indicating the values of different shades of green. The highest cell population and total grid population are given above each heatmap. The blue square indicates a 3×3 action unit that adds population.

4.2.3 BioGymWorld

The BioGymWorld component serves as the interface between the BioEnvironment, Renderer, and the RL agent, and contains the logic for interaction between these. The component inherits from the **gymnasium.Env** class in the **Gymnasium** library. Gymnasium is a Python library for creating RL environments with a standardized API (*Gymnasium* 2023). It builds upon the **OpenAI gym** library. By creating an RL environment with a standardized API, it becomes compatible with third-party RL algorithm libraries such as SB3, which is used in this thesis (more details in section 4.3).

When the RL agent has chosen an action to perform, it sends it to the BioGymWorld component. BioGymWorld passes the action on to the BioEnvironment component, which applies the action and simulates a time step in the wildlife environment. It then returns its current state to BioGymWorld, which passes it, and a reward, on to the RL agent. The current state is also sent to the Renderer, which renders a graphical representation of it. If BioEnvironment reaches a ter-

minal state or the episode ends, BioGymWorld signals this to the RL agent and resets the BioEnvironment and Renderer.

BioGymWorld is also responsible for calculating the reward that the RL agent receives at each time step. The reward function is described in detail in the following subsection.

Reward Function

As the research goal of this thesis states, the goal of the RL agent is to keep a diverse and stable ecosystem through spatio-temporal wildlife management. With this in mind, *critical threshold* was defined as a simple metric of biodiversity on which to base the reward function. A critical threshold is a species-specific population threshold, meaning that each species has its own critical threshold. If a species population is below its critical threshold it is considered to be in a critical state, meaning that the risk of it becoming extinct is higher. Therefore, it is desirable to have populations above their critical threshold. The reward is based on whether or not any species populations are below their respective critical threshold. The critical threshold for a species is defined by the following equation:

$$crit\_thresh_i = extinct\_thresh_i \times grid\_size^2 \times species\_thresh_i \tag{4.6}$$

where $crit\_thresh_i$ is the critical threshold for species i, $extinct\_thresh_i$ is the extinction threshold for species i, $grid\_size$ is the size of the BioEnvironment grid given as the number of rows/columns, and $species\_thresh_i$ is a species-specific threshold multiplier. All of these variables are hyperparameters of the wildlife simulation, and the critical thresholds are therefore set implicitly by the system. For example, let us say we have a BioEnvironment with a grid size of 9 where each cell has a carrying capacity for prey (K) of 100 and a species-specific threshold of 5 for prey. The extinction threshold would then be 5, leading to the critical threshold for prey becoming 2025. In this example, if the total prey population in the whole grid is less than 2025, it would be considered as being critically low.

The equation for critical threshold can be interpreted as each cell in the BioEnvironment on average containing a population equal to $extinct\_thresh_i \times species\_thresh_i$ for species i.

Figure 4.7 shows how species populations might vary to be above or below their critical thresholds over time.

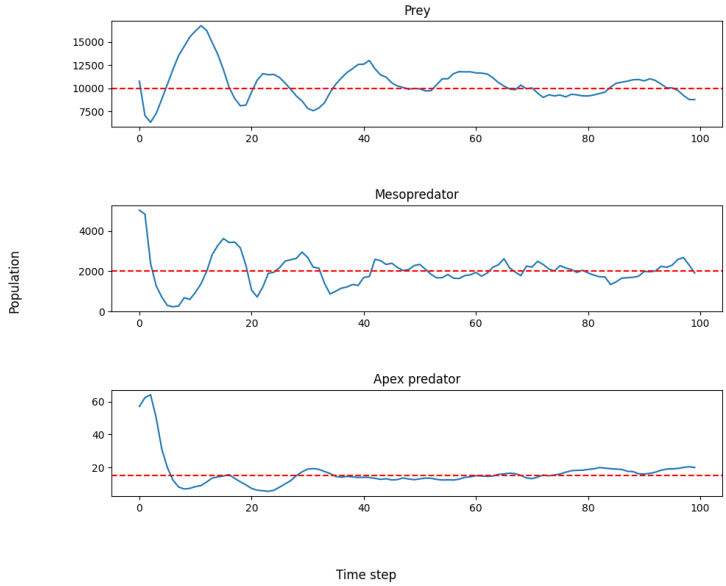


Figure 4.7: Species populations over time in one episode, with the total population in the environment on the y-axis and time step on the x-axis. Critical thresholds are represented as dashed red lines. The agent receives a negative reward if one or more species population is below its critical threshold.

If one or more species have a population below their critical threshold, -1 is added to the total reward. If no species is considered to be in a critical state, 1 is added to the total reward. If a species becomes completely extinct, meaning that its total population in the whole grid is 0, the episode is terminated and a reward of -1000 is given to the agent.

The final reward function also takes into account the practical cost of removing or adding new individuals of a species to the ecosystem. To imitate this cost, placing an action unit negatively influences the total reward. If the action unit adds population, *add_pop_cost* is added to the total reward. This term is defined by the following equation:

$$add\_pop\_cost = -\frac{action\_unit\_size^2 \times action\_multiplier}{200} \tag{4.7}$$

where *action_unit_size* is the size of the action unit in terms of the number

of rows/columns, and *action_multiplier* is the hyperparameter influencing the number of individuals added to each cell in the area covered by the action unit. *add_pop_cost* decreases as the size of the action unit grows and as the action multiplier grows, indicating the higher cost of adding more individuals.

If the action unit removes individuals, *remove_pop_cost* is added to the total reward:

$$remove\_pop\_cost_i = -\frac{action\_unit\_size^2 \times multiplier_i}{200} \tag{4.8}$$

where $multiplier_i$ is given as:

$$multiplier_i = \frac{remove\_pop}{action\_unit\_size^2 \times extinct\_thresh_i}$$

where *remove_pop* is the number of individuals of species i removed. This cost function accounts for the size of the action unit, and the number of individuals removed relative to that species' extinction threshold. The more individuals that need to be removed, the higher the cost, and the less the total reward. Both cost functions are multiplied with $\frac{1}{200}$ to scale them to, in most cases, be larger than -1. This was done to signal that the cost of one or more species being in a critical state, is higher than the practical cost of adding/removing population in most cases. This is because a species in a critical state might potentially go extinct, which would be deemed disastrous.

By combining the reward determined by the critical threshold with the cost functions presented, the final reward function becomes:

$$reward = \begin{cases} -1000 & \text{if } num\_species\_extinct() > 0 \\ -1 + cost & \text{if } num\_species\_critical() > 0 \\ 1 + cost, & \text{otherwise} \end{cases} \tag{4.9}$$

where the *cost* function is defined as:

$$cost = \begin{cases} add\_pop\_cost & \text{if } action\_unit\_add() \\ remove\_pop\_cost_i & \text{if } action\_unit\_remove() \\ 0, & \text{otherwise} \end{cases} \tag{4.10}$$

num_species_extinct() and *num_species_critical*() are functions of the simulations returning the number of extinct species and the number of species in a critical state, respectively. *action_unit_add*() and *action_unit_remove*() are functions returning **True** if the action unit added population or if it removed population, respectively. If no action unit is placed, no cost is added to the total reward.

Biodiversity Metrics

There are different ways to measure the biodiversity of an ecosystem. The BioGymWorld component tracks two such metrics: *species abundance* and *Shannon index*.

Species abundance generally refers to the number of individuals of a species in the ecosystem. In this thesis, it was deemed preferable to represent this metric as one number, as it makes it easier to monitor across time. Therefore, species abundance is defined as the average abundance over all three species. Since different species vary greatly in population sizes, they are each scaled towards their respective critical threshold, resulting in a metric defined in this thesis as *criticalness*. The criticalness of a species is defined as the total species population divided by the species' critical threshold. If this value is less than 1, the species is in a critical state. The species abundance is the average criticalness taken over all three species.

While species abundance gives information about the population sizes in the ecosystem, it does not give any information about the balance of population sizes across species. For example, given a species abundance, it is not possible to know if the three species have relatively equal criticalness (abundance relative to critical threshold), or if there is a big imbalance between criticalness resulting in the same average.

The term *species evenness* is used for describing how close the different species populations are in size compared to each other. One common biodiversity metric that accounts for species evenness is the *Shannon index*[1] (Spellerberg and Fedor, 2003). The Shannon index is given by the following equation:

$$H = -\sum_{i=1}^{S} p_i \ln(p_i) \tag{4.11}$$

[1]The Shannon index is also known as *Shannon entropy* as it was first put forward by Claude Shannon as a measurement of entropy in information theory (Shannon, 1948).

where H is the species diversity, S represents the number of different species in the ecosystem, and p_i is the fraction that the population size of species i makes of the whole ecosystem population. The higher H is, the closer the population sizes of the different species are to each other, and the more diverse the ecosystem is. If one or more species are extinct, H is zero.

While the reward the RL agent receives is used as a proxy indicator for how diverse and stable the ecosystem is, the biodiversity metrics are direct measurements of the biodiversity in the ecosystem. They are therefore useful for determining the true performance of the RL agent on the task of spatio-temporal wildlife management.

Observations

In addition to a reward, BioGymWorld also sends the state of the RL environment to the RL agent, as part of the observations. The state of the RL environment consists of the state of the BioEnvironment, the criticalness for each species, and the criticalness trend for each species.

The state of the BioEnvironment is simply the population grids of each species. These give the raw information about the BioEnvironment at a given time step. It is beneficial for neural networks to have small value inputs, usually, this is done by normalizing the input values. However, initial tests found that scaling the population sizes in all cells toward the critical threshold of the given species gave better learning. Since there are three species in the ecosystem, the state of the BioEnvironment can be represented by $3 \times grid\_size^2$ numerical values.

While it is possible to define the state of the RL environment simply as the state of the BioEnvironment, it can be beneficial to extract some additional features of the state and send them to the RL agent to make it easier for the neural network function approximators to learn the relation between actions, states, and rewards. Some small-scale testing indicated species criticalness and criticalness trends to be good features of the BioEnvironment state. This seems intuitive as the reward that the RL agent receives is directly dependent on whether or not all species are above their critical threshold (criticalness is above 1). The criticalness of a species is defined in the previous subsection.

The criticalness trend for each species is also sent to the agent. It is defined as the change of criticalness from the previous time step and is thus defined as the difference between the current time step criticalness and the previous time step criticalness. The intention behind this feature was to give the RL agent some temporal information about how the BioEnviroment state develops.

The criticalness and criticalness trend of each species is sent as part of the RL environment state, meaning that six additional numerical values are sent to the agent.

4.2.4 Environment Parameters

The spatio-temporal environment can be modified through numerous parameters, which allows for a high variety of simulation behaviors. Therefore, it was necessary to spend some time running through different system configurations to find reasonable parameters. Reasonable parameters refer to parameters that lead to a stable environment where species don't go extinct too often, but there still remains a small probability of it. This is obviously not a clearly defined requirement as there are multiple ways of interpreting what "too often" or "small probability" is. However, the parameter values presented in table 4.1 were found to create a dynamic and challenging environment for the RL agent.

Table 4.1: Spatio-temporal environment parameters.

Parameter	**Description**	**Value**
grid_size	Width and height of the BioEnvironment grid, given as the number of cells.	9×9
action_unit_size	Width and height of the BioEnvironment action unit, given as the number of cells.	2×2 / 3×3 / 4×4
action_multiplier	Constant multiplied with extinction threshold to decide the number of individuals added to each cell when taking an action to add population.	5 / 10 / 15
max_steps	Max steps in an episode	100
species_thresh	Species-specific threshold multiplier affecting the species' critical threshold. Given on the format [Prey, Mesopredator, Apex predator]	[5, 5.5, 5]
migration_rate	The rate of migration out from a cell, for each species. Given on the format [Prey, Mesopredator, Apex predator]	[0.05, 0.05, 0.05]
species_ranges	The range in which a species' population is initialized in a cell. Given on the format [Prey, Mesopredator, Apex predator]	[[0, 18], [0, 6], [0, 0.07]]

The tri-trophic system parameter values used in all experiments are presented in table 4.2. These were based on parameter values presented in Fryxell (2014, p. 168) of a wolf-moose-plant ecosystem. Extinction thresholds and critical thresholds are set implicitly based on these parameters, as described earlier in this section.

Table 4.2: Tri-trophic system parameter values used in experiments.

Parameter	**Description**	**Value**
r_{max}	Maximum reproduction per prey	3.33
K	Carrying capacity of prey in a cell	70
a	Rate of prey consumption by a mesopredator	2
b	The number at which the mesopredator consumption of the prey is half of its maximum	40
e	Conversion of prey consumption to mesopredator offspring	2.1
d	Decrease in the mesopredator population due to natural reasons such as death	0.7
A	Rate of mesopredator consumption by a apex predator	12.3
B	The number at which the apex predator consumption of the mesopredator is half of its maximum	0.47
E	Conversion of mesopredator consumption to apex predator offspring	0.1
D	Decrease in the apex predator population due to natural reasons such as death	0.45
s	Maximum rate of apex predators per capita	0.4
γ	Maximum apex predator density	0.1

To answer research question 1, which aims to find the best action set in terms of performance, a set of values for the action_unit_size parameter and a set of values for the action_multiplier parameter was decided upon. These are presented in table 4.1. These sets of candidate values were found through small-scale testing of the environment. For action units larger than 4×4, the impact of each action seemed to be too significant and the environment quickly became imbalanced, which often lead to one or more species becoming extinct. On the other hand,

if the action unit is the size of one cell, the impact of each action seems to be too small to make any meaningful difference. The same observations applied to action_multiplier, where the values between 5 and 15 seemed to give the agent a reasonable amount of impact with each action. Three potential values were selected for each parameter to be tested, as it seemed reasonable to achieve valuable results with this number of values within the time frame of this thesis.

4.3 RL Algorithms

An RL agent is trained by following an RL algorithm. The aim of the RL algorithm is to learn a good value function or policy function which takes actions that maximizes the reward the agent receives. Research question 2 in this thesis was posed to explore which of the following DRL algorithms perform best when it comes to that task: DQN, A2C, or PPO. All of the algorithms have implementations in the *Stable Baselines3* (SB3) library (Raffin et al., 2021). SB3 is a library of common, deep RL algorithm implementations in Python. The algorithms are implemented in the machine learning framework *PyTorch*.

There are several advantages of using an RL library, such as SB3, over implementing the algorithms from scratch. Using SB3 ensures stable RL algorithm implementations of high quality, as it is developed and maintained by a large group of RL researchers. Implementing the algorithms from scratch would, in addition to being time-consuming, be prone to flaws and bugs. Using SB3, one can be more confident that performance differences between RL algorithms are due to underlying theoretical differences, and less because of differences in implementation.

The SB3 implementations of DQN, A2C, and PPO were tested on the spatio-temporal wildlife environment. Specific implementation details are presented in the following subsections.

4.3.1 DQN

The SB3 implementation of DQN is based on the updated algorithm, which utilized a target network. This algorithm was proposed by Mnih et al. (2015) and is described in section 3.4.1. An overview of the architecture is shown in figure 3.4.

The default SB3 hyperparameters for DQN were used, and are given in table 4.3.

Table 4.3: DQN hyperparameters used in experiments.

Hyperparameter	**Description**	**Value**
learning_rate	Learning rate for Q-network	0.0001
buffer_size	Replay memory size	1 000 000
learning_starts	Number of steps where the agent collects experiences before starting learning	50 000
batch_size	Minibatch size for each gradient update	32
gamma	The discount factor (γ)	0.99
train_freq	Q-network update frequency (number of steps)	4
gradient_steps	Number of gradient steps performed on a minibatch	1
target_update_interval	Number of steps between each target network update	10 000
exploration_fraction	Fraction of the training period where the exploration rate decreases	0.1
exploration_initial_eps	Initial probability of random action	1.0
exploration_final_eps	Final probability of random action	0.05

This DQN implementation has two phases, and the hyperparameter learning_starts in table 4.3 defines the transition between these two. The first phase of training this DQN algorithm consists solely of collecting experiences in the replay memory, and it does not update its Q-network. This phase is important as a larger number of experiences in the replay memory lowers the risk of samples in a minibatch having a temporal correlation. Once the replay memory has gained a fairly large number of experiences (exact number given by learning_starts), it begins to learn by sampling minibatches of experiences from the replay memory and updating the Q-network and target network.

The SB3 default Q-network was used in experiments. It was a fully connected deep neural network with two hidden layers. Each layer consisted of 64 nodes and used the **ReLU** activation function. The target network was identical to the Q-network.

4.3.2 A2C

The A2C algorithm described in section 3.4.2 proposed having multiple instances of the RL environment in a vectorized environment. In SB3 this is made possible by "wrapping" your custom environment in the **SubprocVecEnv** wrapper. **SubprocVecEnv** creates an environment consisting of multiple instances of that environment. Each instance of the environment runs on its own process on the computer, allowing for multiple instances to run simultaneously. After wrapping the BioGymWorld class in the **SubprocVecEnv** wrapper, it was wrapped in the **VecMonitor** wrapper (also available in SB3). **VecMonitor** monitors and saves useful information about the vectorized environment when training, such as the average episode reward.

As the experiments were run on an 8-core CPU (Apple M1), the vectorized environment was set to contain eight environment instances. This way each environment instance, which is initialized as its own process, can run on its own core.

For the A2C implementation used in this thesis, default SB3 hyperparameters were used. Some of the relevant hyperparameters are presented in table 4.4.

Table 4.4: A2C hyperparameters used in experiments.

Hyperparameter	**Description**	**Value**
learning_rate	Learning rate for the shared actor and critic network	0.0007
n_steps	Number of steps each environment runs for each update	5
gamma	The discount factor (γ)	0.99

As described in section 3.4.2, the actor and critic share all layers of a single neural network except for the output layer. The SB3 default neural network was used, which consists of 2 fully connected hidden layers with 64 nodes each. The **tanh** activation function was used.

4.3.3 PPO

The SB3 implementation of PPO is similar to the A2C implementation, with the main difference being the "clipped" surrogate loss function described in section 3.4.3. The PPO algorithm is also set to contain eight environment instances

that run in parallel. The default SB3 hyperparameters were used in experiments. Relevant hyperparameters are described in table 4.5.

Table 4.5: PPO hyperparameters used in experiments.

Hyperparameter	**Description**	**Value**
learning_rate	Learning rate for the shared actor and critic network	0.0003
n_steps	Number of steps each environment runs for each update	2048
batch_size	Minibatch size for training the network	64
n_epochs	Number of epochs for optimizing surrogate loss	10
gamma	The discount factor (γ)	0.99
clip_range	The clipping parameter (ϵ)	0.2
normalize_advantage	Whether or not to normalize the advantage	True
vf_coef	Value function coefficient (c_1)	0.5
ent_coef	Entropy coefficient (c_2)	0

Notice that the entropy coefficient is set to zero, meaning that the entropy term in equation 3.11 disappears.

The neural network shared between the actor and the critic is identical to the one used in the A2C implementation (details in the preceding subsection).

4.4 Experimental Plan

To answer the two research questions posed in this thesis, an experimental plan was made and followed.

Research question 1 focuses on finding the action set that enables the best performance. To determine this, the best combination of values for the action unit size and action multiplier parameters needs to be found. There are three potential values for each of these parameters. These are presented in table 4.1, and the justification for their selection is in subsection 4.2.4. To evaluate the performance of a specific action set, one could look at how well the different DRL algorithms are able to perform with it. The performance of an algorithm can be evaluated by looking at the average episode reward they receive.

Environment stochasticity and the inherent randomness in learning can greatly affect the performance of DRL algorithms (Henderson et al., 2019). Therefore, it can be beneficial to train and evaluate an algorithm multiple times with the same action set i.e. have multiple *runs* to reduce the effect of this stochasticity on the average rewards and be able to draw more robust conclusions. To this end, each algorithm was run 20 times with each specific action set.

Each DRL algorithm was trained for approximately 200 000 RL environment steps as initial testing suggested that this was enough data for most of the algorithms to converge with regard to the average episode reward they achieve[2].

It would be possible to test all possible combinations of values for the two parameters affecting the action set. However, since there are 9 possible combinations, each being tested 20 times on all three DRL algorithms, this would result in 540 runs. Given the hardware available for training the algorithms, this was deemed a too time-consuming and resource-intensive approach. It was therefore decided to do this in an alternative, more systematic manner which would reduce the total number of runs: The DRL algorithms were first trained and evaluated on action sets with varying action unit sizes, but a fixed action multiplier. Based on these results, one could get an indication of what the best action unit size is. The DRL algorithms were then trained and evaluated on action sets with the best action unit size but with varying action multipliers. Based on the performance of the algorithms, one could determine the best action multiplier. The best action unit size and action multiplier would make up the best action set, thus answering research question 1.

To answer research question 2, one would have to look at the performance of the different DRL algorithms. However, in addition to comparing the algorithms with regard to the average episode rewards, they were also compared to each other with regard to additional biodiversity metrics.

The parameters used for the algorithms and the wildlife environment simulation in the experiments are presented in the preceding sections.

[2]While both DQN and A2C train for exactly 200 000 time steps, PPO was trained for 212 992 time steps. The reason behind this is that the *n_steps* hyperparameter is set to 2048 in the PPO implementation. As the PPO is run with 8 parallel environment instances in this thesis, it essentially collects batches of $2048 \times 8 = 16384$ steps. PPO trains for 212 992 steps as it is the first multiple of 16 384 over 200 000.

4.5 Summary

In summary, this chapter has described how a system consisting of an RL environment and DRL algorithms was constructed, along with the experimental plan designed to generate results to answer the research questions of this thesis. An overview of the system was given in section 4.1. The system consists of an RL environment referred to as the spatio-temporal wildlife environment and the DRL algorithms tested on it. The wildlife environment was built from scratch and a detailed description of its components was given in section 4.2. The DRL algorithms were provided by the RL library Stable Baselines3, and implementation details regarding these are presented in section 4.3. Finally, section 4.4 lays out the experiments performed on this system to yield the results presented in the subsequent chapter.

Chapter 5

Results and Discussion

This chapter presents the results obtained to answer the research questions of this thesis and thoroughly discusses them. Section 5.1 show the results obtained by testing the different DRL algorithms with different action sets. These results are then discussed in more detail in section 5.2. Section 5.3 serves as a summary of the main results and the central discussion points in this chapter.

5.1 Results

The results presented in this subsection were obtained by following the experimental plan presented in section 4.4. The initial experiments focused on finding the best action unit size. This was done with a fixed action multiplier value of 10x, which is one of the candidate action multiplier values, but also the mean (and median) of all candidate values. Each algorithm was trained 20 times for approximately 200 000 time steps in the RL environment on each action unit size. The mean of the 20 runs as well as the 95% confidence interval was then calculated and plotted.

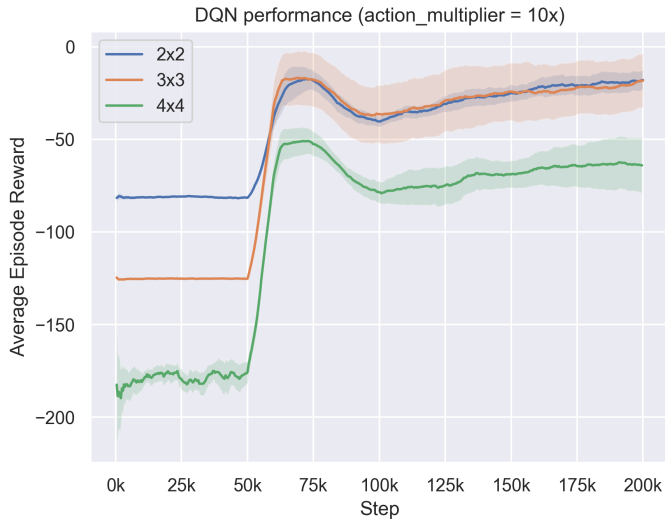


Figure 5.1: The performance of the DQN algorithm during training with different action unit sizes and with action multiplier set to 10x. The y-axis shows the average episode reward. The x-axis displays the number of RL environment time steps the algorithm has trained for. The algorithm was trained 20 times for 200 000 time steps in the RL environment for each action unit size. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

Figure 5.1 shows the results of training the DQN algorithm with a 10x action multiplier and varying action unit sizes. The results show that the runs where the algorithm is trained on 2×2 and 3×3 action unit sizes obtain approximately the same average episode reward of -18 at the end of training. However, when the algorithm is trained on 3×3 action unit size it has a larger confidence interval, indicating more variability between its runs. When the algorithm is trained on a 4×4 action unit size it performs worse than the aforementioned two, with it achieving an average reward of -64 at the end of its training runs.

The results show that the DQN algorithm does not improve its average episode reward for the first 50 000 steps. This is because the hyperparameter learning_starts was set to 50 000 for the implementation used, as presented in table 4.3. As described in subsection 4.3.1, this means that the algorithm only collects experiences to its replay memory in this phase, hence why the performance does

not change notably. In the second phase, after 50 000 time steps, the algorithm begins to learn by sampling minibatches of experiences from the replay memory and updating its Q-network and target network. As the results show, learning significantly improves performance, indicating that the Q-network becomes better at estimating Q-values for this RL environment.

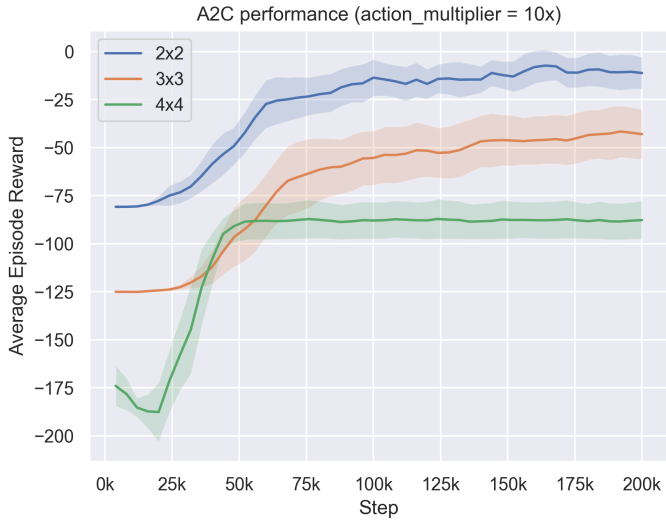


Figure 5.2: The performance of the A2C algorithm (running 8 parallel environments) during training with different action unit sizes and with action multiplier set to 10x. The y-axis shows the average episode reward. The x-axis displays the number of RL environment time steps the algorithm has trained for. The algorithm was trained 20 times for 200 000 time steps in the RL environment for each action unit size. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

Figure 5.2 displays the results of training the A2C algorithm with a 10x action multiplier and varying action unit sizes. The algorithm was trained with 8 different environment instances running in parallel at the same time. The algorithm trained with 2×2 action unit size performs the best with an average episode reward of -11 at the end of its training runs. The second-best performance is achieved by the algorithm trained with the 3×3 action unit size, with an average reward of -43 at the end of training. The algorithm trained with 4×4 action unit size performs the worst. It converges early to an average reward of -88.

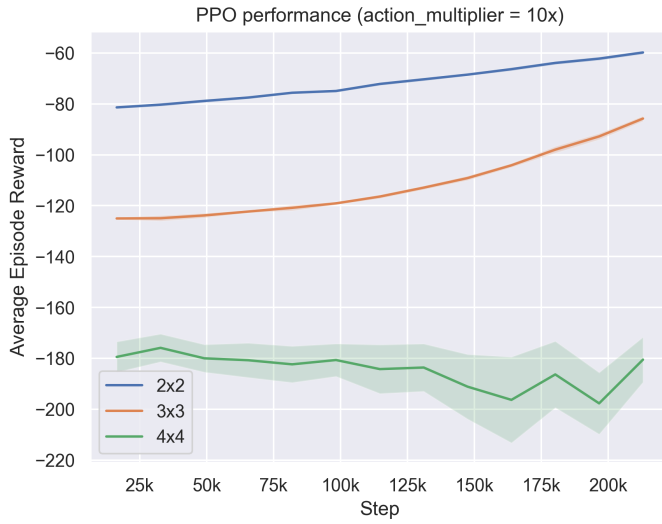


Figure 5.3: The performance of the PPO algorithm (running 8 parallel environments) during training with different action unit sizes and with action multiplier set to 10x. The y-axis shows the average episode reward. The x-axis displays the number of RL environment time steps the algorithm has trained for. The algorithm was trained 20 times for 212 992 time steps in the RL environment for each action unit size. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

The results of training the PPO algorithm on a 10x action multiplier and different action unit sizes are shown in figure 5.3. The algorithm trained with the 2×2 action unit size yields the best performance. It receives an average episode reward of -60 at the end of training. The algorithm trained with the 3×3 action unit size is second-best and receives an average episode reward of -86 after training. Similar to the other two DRL algorithms tested with a 10x action multiplier, training with a 4×4 action unit size results in the worst performance for the PPO algorithm. After 212 992 training steps, it earns an average episode reward of -180, thus not becoming any better than when it started training.

While training with 4×4 action unit size showed variability between runs, the 2×2 and 3×3 action unit sizes showed virtually no variability, as shown by the results.

The results of running the different DRL algorithms with an action multiplier of 10x and with varying action unit sizes show that the 2×2 action unit size performs either best, as observed with A2C and PPO, or on par with the best, as in the case of DQN. These findings indicate that the 2×2 action unit size might be best for all DRL algorithms trained on the spatio-temporal wildlife environment. Therefore, it was decided to use this action unit size to find the best action multiplier.

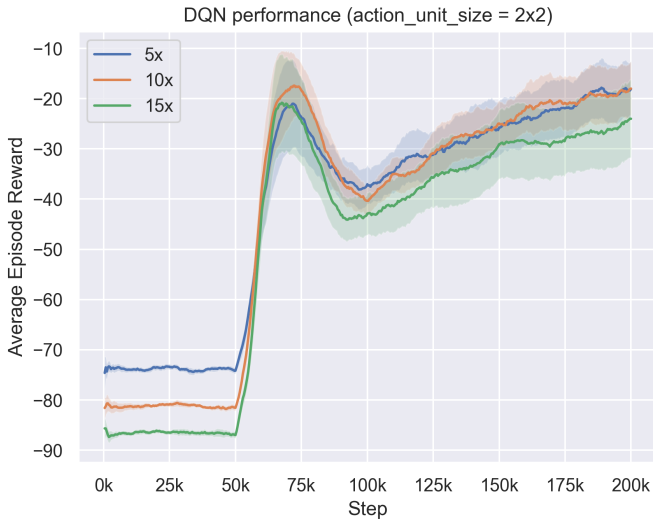


Figure 5.4: The performance of the DQN algorithm during training with different action multipliers and with action unit size set to 2×2. The y-axis shows the average episode reward. The x-axis displays the number of RL environment time steps the algorithm has trained for. The algorithm was trained 20 times for 200 000 time steps in the RL environment for each action multiplier value. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

Figure 5.4 shows the results of training the DQN algorithm with 2×2 action unit size and varying action multipliers. Both the algorithm trained with a 5x action multiplier and the one trained with a 10x action multiplier earns an average episode reward of -18. The DQN algorithm trained with a 15x action multiplier performs the worst, with an average episode reward of -24 after training.

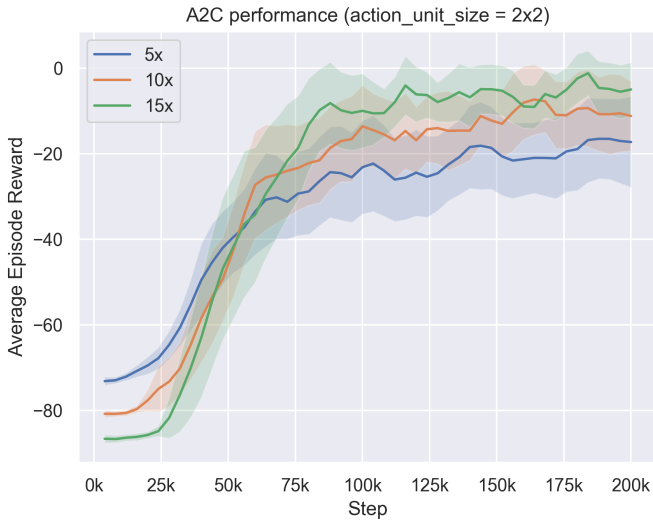


Figure 5.5: The performance of the A2C algorithm (running 8 parallel environments) during training with different action multipliers and with action unit size set to 2×2. The y-axis shows the average episode reward. The x-axis displays the number of RL environment time steps the algorithm has trained for. The algorithm was trained 20 times for 200 000 time steps in the RL environment for each action multiplier value. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

The performance of the A2C algorithm with a 2×2 action unit size and different action multipliers is shown in figure 5.5. Contrary to the results achieved with DQN, the A2C algorithm trained with a 15x action multiplier performs best, receiving an average episode reward of -5. The algorithm trained with 10x is the second-best after training. It earns an average episode reward of -11. The algorithm trained with an action multiplier of 5x receives an average episode reward of -17 and thus becomes the worst-performing one.

Figure 5.6: The performance of the PPO algorithm (running 8 parallel environments) during training with different action multipliers and with action unit size set to 2×2. The y-axis shows the average episode reward. The x-axis displays the number of RL environment time steps the algorithm has trained for. The algorithm was trained 20 times for 212 992 time steps in the RL environment for each action multiplier value. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

Figure 5.6 displays the results of training the PPO algorithm with 2×2 action unit size and varying action multipliers. The algorithm trained with a 5x action multiplier achieves an average episode reward of -51 after 212 992 time steps, making it the best-performing one. Its performance is followed by the algorithm trained with a 10x action multiplier, which earns an average episode reward of -60 at the end of training. Similarly to DQN, but contrary to A2C, the PPO algorithm trained with a 15x action multiplier performs the worst. It yields an average episode reward of -67.

The results of training the different DRL algorithms with a 2×2 action unit size show that there is no single action multiplier that is best for all algorithms. While the DQN algorithm seems to perform best with smaller action multipliers, A2C achieves better results the larger the action multiplier is. Similarly to DQN, PPO seems to perform better with a smaller action multiplier. However, in contrast to

the two other algorithms, the PPO results show steady but rapid improvement throughout the whole training. This strongly indicates that the algorithm has not reached its best performance and that it would significantly improve had it been trained for longer.

To investigate this further, one could compare performance over RL environment steps with performance over wall clock time. To do this it was decided to use the action set consisting of a 2×2 action unit size and a 10x action multiplier. The 2×2 action unit size seems to be the best action unit size based on the initial experiments. The latter experiments indicate that there is no clear best action multiplier across the different DRL algorithms. However, the 10x action multiplier is the only one to show average performance or even best performance in the case of DQN, across all algorithms tested with a 2×2 action unit size. It was therefore decided to use it as part of the action set in these experiments.

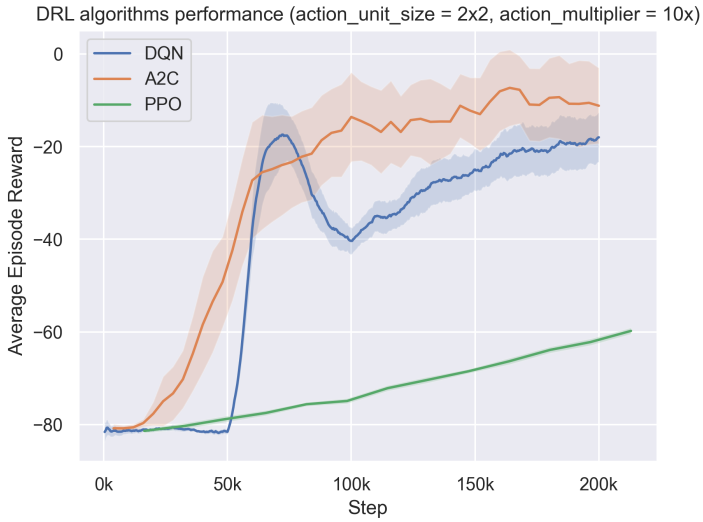


Figure 5.7: The performance of DQN, A2C and PPO during training with 2×2 action unit size and action multiplier set to 10x. The y-axis shows the average episode reward. The x-axis displays the number of RL environment time steps the algorithm has trained for. The algorithms were trained 20 times for approximately 200 000 time steps in the RL environment. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

Figure 5.7 display the results of training the different DRL algorithms with 2×2 action unit size and 10x action multiplier. The results show that the A2C and DQN algorithms rapidly improve the first 60 000 steps before slowly converging to an average episode reward between -10 and -20. The performance of the PPO algorithm in this figure seems stable but slow, with it ultimately ending training with an average episode reward of -60.

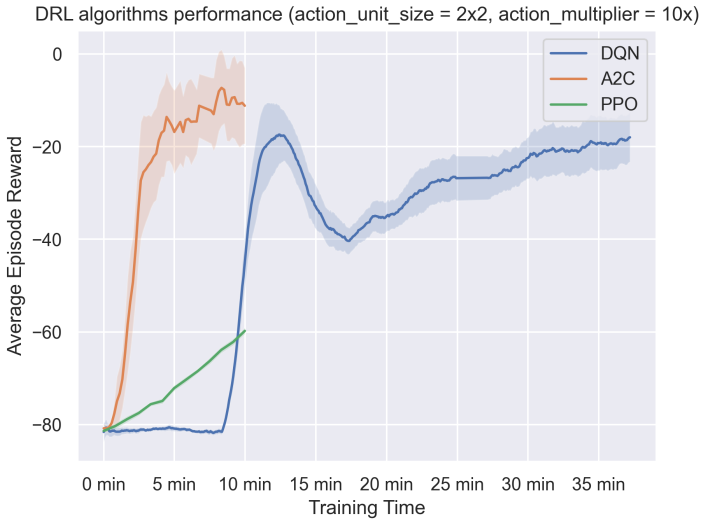


Figure 5.8: The performance of DQN, A2C and PPO during training with 2×2 action unit size and action multiplier set to 10x. The y-axis shows the average episode reward. The x-axis displays the wall clock time the algorithm has trained for. The algorithms were trained 20 times for approximately 200 000 time steps in the RL environment. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

Figure 5.8 shows an alternate way of displaying the results in figure 5.7. In this figure, the x-axis displays the training time instead of the number of RL environment steps trained over. Training time in this case refers to the wall clock time, meaning the real-life time that has passed during training. Since both A2C and PPO run multiple environments in parallel, they take RL environment steps at a much faster rate than DQN. This is reflected in this figure, where both A2C and PPO take roughly 10 minutes to train over approximately 200 000 environment steps, while DQN uses 37 minutes. In other words, training A2C

and PPO is approximately 3.7 times faster than DQN with regard to training time[1].

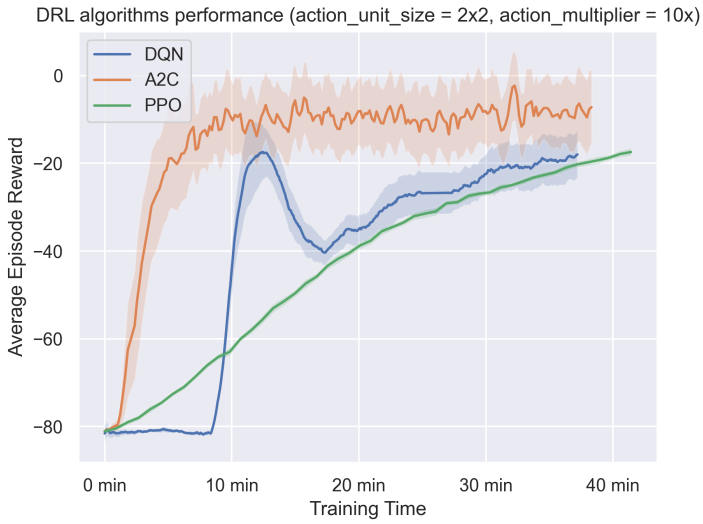


Figure 5.9: The performance of DQN, A2C and PPO during training with 2×2 action unit size and action multiplier set to 10x. The y-axis shows the average episode reward. The x-axis displays the wall clock time the algorithm has trained for. The algorithms were trained 20 times for approximately 37 minutes (200 000 time steps for DQN, 800 000 time steps for A2C, and 802 816 time steps for PPO) in the RL environment. The mean of these 20 runs was then plotted, with the shading representing the related 95% confidence interval.

Given that A2C and PPO are training considerably faster than DQN with regard to wall clock time, it would be interesting to view how well they perform if they train for the same amount of time as DQN. Figure 5.9 shows the performance of the DRL algorithms when they are trained for approximately the same amount of wall clock time. As A2C and PPO are roughly 3.7 times faster than DQN, they were trained for 4 times more RL environment steps than DQN. The results indicate that the performance of A2C does not improve notably after 10 minutes

[1]As both A2C and PPO run 8 parallel environment instances it is natural to assume that training would be 8 times faster than with DQN. However, each environment instance is its own operating system process, and overhead introduced by process communication makes the practical speed-up smaller.

(200 000 time steps). The 95% confidence interval (shaded area) displays its relatively high instability in performance. The DQN algorithm also reaches its peak performance after around 10 minutes (70 000 time steps), however, after a dip in performance, its average episode reward increases slowly until the end of training. The PPO algorithm displays a slow, stable, and consistent increase in average episode reward throughout the training duration. Toward the end of the training, the difference in performance between the DRL algorithms is relatively small.

Even though the different DRL algorithms perform relatively similarly after a set amount of time, they employ considerably different policies. This is reflected in table 5.1, which shows the percentage of different types of actions taken by the algorithms.

Table 5.1: Percentage of the different types of action units placed for the DRL algorithms trained with a 2×2 action unit size and a 10x action multiplier over approximately 37 minutes in the RL environment. The percentages are calculated by running 50 episodes of the best run for each algorithm.

DRL algorithm	**% adding population**	**% removing population**	**% no action unit placed**
DQN	8.16	91.84	0
A2C	0	100	0
PPO	90.14	9.86	0

The table shows that the DQN algorithm trains the RL agent to mainly place action units that remove species population, while PPO trains the agent to predominantly place action units adding population. Training the RL agent with the A2C algorithm makes it follow a simple policy: always remove species population. Figures A.1, A.2 and A.3 in appendix A offer more detailed examples of the policies, showing sequences of actions taken by the agent when trained with the different DRL algorithms.

Table 5.2: Average species abundance and Shannon index and their standard deviation for the DRL algorithms trained with a 2×2 action unit size and a 10x action multiplier over approximately 37 minutes in the RL environment. The average and standard deviation are calculated by running 50 episodes of the best run for each algorithm.

DRL algorithm	Species abundance	Shannon index
DQN	1.321 ± 0.018	0.513 ± 0.005
A2C	1.314 ± 0.022	0.506 ± 0.007
PPO	1.341 ± 0.014	0.529 ± 0.005

Table 5.2 shows how the different DRL algorithms score on the biodiversity metrics described in section 4.2.3. PPO scores highest on both species abundance and Shannon index, with DQN scoring second highest on both. A2C scores the lowest and has the highest standard deviation for both metrics. This aligns with the simple policy of A2C of only removing populations of a specific species in the same place.

5.2 Discussion

The previous section presented the results of training the DRL algorithms with different action sets and for varying duration. The results displayed how all the DRL algorithms performed best with the 2×2 action unit size but had different best action multipliers given this action unit size. The results also displayed the difference in algorithm performance during training over a fixed number of RL environment steps, compared to a fixed amount of wall clock time. This section takes a deeper look at the results and some of the underlying patterns found, while also discussing their potential causal factors.

5.2.1 Finding the Best Action Set

As explained in the experimental plan for this thesis (section 4.4), a systematic way of collecting results was followed to find the best action set. This significantly reduced the number of runs needed to collect results. However, this also meant that not all combinations of action unit sizes and action multipliers were tested. One can therefore not conclude with certainty that the best action set has been found for each DRL algorithm. The results do however indicate that all the

algorithms perform best with the smallest action unit size, 2×2. There could be multiple reasons for this. The most obvious reason is that smaller action unit sizes add fewer individuals, thus having less cost per action. Because of the structure of the reward function, less cost means less negative reward added to the total reward the RL agent receives. This effect on the reward is apparent in results comparing performance with different action units, where at the start of training, the algorithms training with smaller action unit sizes, begin training with a higher average episode reward. This effect also applies to action multipliers, as a lower action multiplier corresponds to fewer individuals of a species being added to the environment. However, less cost also means fewer individuals added to the ecosystem. Therefore the performance of an algorithm cannot solely be attributed to the amount of cost associated with its action set. This is demonstrated by figures 5.1, 5.4, and 5.5, which display algorithms trained with a higher cost action set performing as good or better than algorithms trained with the action set with the lowest cost. This implies that the action set clearly affects the ability of the algorithm to perform well in the RL environment, beyond its costs. If an action set is efficiently used, it might make up for the higher cost compared to the lowest-cost action set. If it did not, the action set with the lowest cost would always yield the highest average episode reward.

While the 2×2 action unit size seems to be best for all algorithms, there is no clear best action multiplier for this action unit size. DQN achieved the best performance with 5x and 10x action multipliers, while A2C got the highest average episode reward with an action multiplier of 15x. PPO yielded its highest average episode reward when trained with an action multiplier of 5x. However, as figure 5.6 displays, the algorithm has clearly not reached its best performance after 200 000 time steps. The final results might have been different, had PPO been trained for longer.

One explanation for why the algorithms have different best action sets is that they employ different policies, as seen in table 5.1. For example, when the RL agent is trained with A2C it always chooses to remove species population. Since the action multiplier does not affect the cost of removing species population, it essentially does not affect the RL agent trained with A2C. However, this contradicts the results shown in figure 5.5, where the agent trained with a higher action multiplier performs better. One hypothesis for this is that the agent trained with a higher action multiplier is more prone to learn a policy of just removing species population. This could be because it relates actions adding population to a larger negative return than an agent trained with a smaller action multiplier, because of the increased cost, thus being less likely to choose them. This hypothesis is built on the assumption that a policy of just removing species populations yields a higher reward than a policy of both adding and removing populations.

For DQN and PPO, which train the RL agent to both remove and add population, a lower action multiplier makes the agent perform better. Just like action unit size, this is probably due to less cost, but also because it allows the agent to operate more efficiently in the environment.

As discussed above, smaller action unit sizes and smaller action multipliers generally seem to make the RL agent operate more efficiently when trained with the different DRL algorithms. One reason for this might be that higher action unit sizes and action multipliers might add too many individuals at the same place and at the same time than what would be natural for the ecosystem. It might be better to add fewer individuals, but multiple times and spread across time and space to not create changes that are too drastic for the ecosystem. The optimal amount of individuals to add with each action naturally depends on, among other factors of the wildlife simulation, the temporal difference between consecutive states in the RL environment. For example, if the RL environment simulates years in the tri-trophic system between consecutive states the action set might need to increase or decrease more individuals with each action than if it had simulated months. To further highlight the influence of temporal difference on the optimal action set one could flip the problem and see it in relation to the common RL problem of choosing a reasonable *frame skip*. Frame skip is a hyperparameter determining the frequency at which the RL agent can take action. In this thesis, the equivalent would be the temporal difference between consecutive states in the spatio-temporal environment. Frame-skip has been shown to be an important hyperparameter for the performance of the RL agent in other problems (Braylan et al., 2015). This further strengthens the notion that the temporal difference between consecutive states and the optimal action set is connected.

5.2.2 Comparing DRL Algorithm Performance

There are multiple ways to compare the performance of DRL algorithms on this problem. One way would be to compare them with their respective best action set, the one they have achieved the highest reward with. However, deciding upon the best action set for an algorithm is not always straightforward. As the results for DQN show in figures 5.1 and 5.4, an algorithm might achieve similar performance with more than one action set. It might also be the case that the action set with which an algorithm performs best might change as training progresses. Figure 5.5 displays such a case. Therefore, to compare on equal ground, it was decided to look at algorithm performance when they were trained with the same action set. It was decided to use the action set consisting of a 2×2 action unit size and a 10x action multiplier. The reason for this was that

all DRL algorithms had been shown to perform best with a 2×2 action unit size and at least average with a 10x action multiplier.

The results in figure 5.7 show how the DRL algorithms perform with the chosen action set being trained over a similar amount of environment time steps. The relatively quick increase in average episode reward for A2C and DQN stands out compared to the slow learning of PPO. The seemingly quick learning of A2C stands out even more when looking at figure 5.8, which show algorithm performance during training over wall clock time rather than environment time steps. In fact, the A2C algorithm achieves a higher average episode reward in 10 minutes than DQN and PPO are able to achieve in roughly 40 minutes, as shown by figure 5.9. There might be numerous reasons behind the remarkable performance of A2C. One important factor could be that learning a good policy in the spatio-temporal wildlife environment is simpler than learning a good state-value function. In such cases, policy-gradient methods, such as A2C, usually outperform RL methods which only learn a value function, such as DQN. The underlying reason why a good policy is easier to learn in the wildlife environment might have to do with its structure. The RL environment in this thesis is a grid representing a spatial environment. Related work has shown actor-critic methods to outperform DQN in grid environments (section 3.4.2).

The notion of a good policy being easier to learn than a good value function gets further strengthened by the fact that the policy A2C learns is very simple. As table 5.1 shows, A2C only places action units removing species population. In fact, the algorithm trains the agent to always remove species population in the same area, as is demonstrated by the action sequence in figure A.2. The overall strategy seems to be to remove mesopredators from a specific ecosystem area, essentially conserving the prey there. The prey can then grow and migrate to neighboring cells in the ecosystem, where they can be consumed by the mesopredator, which in turn are consumed by the apex predators. The related work presented in section 3.2.3 concluded with a similar strategy performing best. In their case, the reduction of the predator species led to short-term stability in the ecosystem. This strategy, although simple, seems to be effective. However, this approach also shows a relatively large variability in performance as shown by the unstable mean and large 95% confidence interval of A2C in figure 5.9. These results suggest that the simple strategy of A2C might not work as well for all the different states of the RL environment.

Although the DQN algorithm does not achieve a higher average episode reward than A2C, it slowly reduces the performance gap as it trains over more environment steps. This suggests that learning a good value function might not be considerably more complicated than learning a good policy. The results in fig-

ure 5.7 also highlight the relatively high sample efficiency of DQN. As DQN is an off-policy RL method, it collects samples through its behavior policy. As the behavior policy remains unchanged during training, samples collected from it can be stored in the replay memory and reused to update the target policy. On-policy methods, such as A2C and PPO, update their policy during training and therefore need to collect new samples according to this updated policy continuously. Reuse of samples is therefore not possible. This sample inefficiency is reflected by the learning curve of PPO in figure 5.9. PPO requires training over roughly four times the number of environment steps DQN requires to achieve similar performance. However, since A2C and PPO train by collecting samples from environment instances running in parallel, they are also training over environment steps considerably faster than DQN. Overall, A2C and PPO compensate for their sample inefficiency by collecting samples at a much higher rate.

While A2C and DQN show rapid improvements in their performance, PPO displays slow and steady learning across the training period. However, it is remarkably stable as its 95% confidence interval is significantly smaller than the other two algorithms. The stability and consistency in PPO runs can be attributed to its clipped "surrogate" objective function. As described in section 3.4.3, the PPO clipped objective function ensures that the new policy of the agent is close to its old. This is the reason for the relatively slow increase in average episode reward during training, compared to A2C. However, this also makes the performance increase during training much more stable than A2C. While A2C and PPO are similar in many ways, it is interesting to see that they learn two very different policies. As seen in table 5.1, A2C only removes population, while PPO mostly adds population. One hypothesis for this is that A2C quickly adopts the simple and seemingly effective strategy of only removing populations, while PPO takes a more cautious approach, that might lead to a better policy. Because the objective function of PPO restricts large updates to its policy, it is less prone to adopt actions it explores and seems good but lead to a suboptimal policy. In other words, PPO might find an equally as good or better policy than A2C in the long run. This hypothesis is supported by the fact that even though the policy of PPO is very different from A2C, the performance difference becomes less throughout training, as shown in figure 5.9. The figure also shows that while A2C stagnates early, PPO continues to improve until the end of training, indicating that it might be able to perform as well or even better than A2C given enough training time.

While PPO does not have the highest average episode reward at the end of training, it does score the highest on both species abundance and Shannon index as seen in table 5.2. It also has the lowest standard deviation on these metrics, meaning that its policy is more robust than that of DQN and A2C. This reveals that while the reward function is designed to keep species abundances from get-

ting critically low, it does not necessarily reward higher biodiversity. The findings also imply that the policy of PPO might be better than the policies of A2C and DQN with regard to biodiversity even though it receives a lower reward. This is probably because of the relatively high cost of adding species population.

Overall, the performance of the different DRL algorithms can be seen in relation to the *No Free Lunch* (NFL) theorem. The NFL theorem was introduced by Wolpert (1996) and it states that there is no single machine learning algorithm that performs best in all circumstances. In other words, different algorithms are suited to different tasks as they make different assumptions about the problem space. For example, policy gradient algorithms try to learn a good policy and are based on the assumption that it is easier to learn than a good value function. The results show that different assumptions lead to a trade-off between the speed of performance improvement during training and training stability. A2C has a rapid increase in reward from the beginning but shows high variability. PPO, on the other hand, assumes that the environment is complex, and does not want to quickly adopt a policy that might be suboptimal. Therefore, learning is slow but very stable. DQN falls in between A2C and PPO when it comes to both training time and stability.

It is important to note that hyperparameters can have a huge influence on the performance of a DRL algorithm. Especially hyperparameters such as learning rate and epsilon (probability of random action) can influence learning and exploration. Hyperparameter tuning is, therefore, a significant part of training a DRL algorithm on an RL task. However, since hyperparameter search is a computationally complex and resource-intensive process, it was out of the scope of this thesis.

5.3 Summary

This chapter has presented the results of the experiments conducted in this thesis and a discussion around these. The results presented in section 5.1 were yielded by training the DRL algorithms DQN, A2C, and PPO on the RL environment with different action sets. The results indicated that a 2×2 action unit size gave the best performance for all algorithms. While there did not seem to be any clear best action multiplier, an action multiplier of 10x seemed to at least yield average performance for all DRL algorithms. Based on these results, further experiments were conducted with an action set consisting of a 2×2 action unit size and a 10x action multiplier. Results from these were presented in two ways: with regard to environment steps trained on and with regard to wall clock training time.

While DQN showed a higher sample efficiency, A2C and PPO compensated by being much faster to train over environment steps. This was expected as A2C and PPO ran multiple environment instances in parallel. While A2C quickly improved performance, it had a high variability between training runs. PPO, on the other hand, had a slow but very stable increase in performance. DQN fell in between A2C and PPO in both performance increase and stability. The results also revealed the vast difference in action policy between the algorithms, and also their performance according to biodiversity metrics.

The discussion in section 5.2 implied that there might be a connection between the best action set and the temporal difference between states in the RL environment. The performance of the DRL algorithms was also discussed. A2C performed remarkably well in a short amount of time, with the main hypothesis for this being that a good policy in this RL environment is easier to learn than a good value function. However, a good Q-function might not be much harder to learn, as the performance of DQN was relatively close. PPO, while not achieving the highest reward, performed best on the biodiversity metrics. It also showed promise as it closed the performance gap with the other algorithms as training time increased. This, in addition to its slow but stable learning, can most likely be attributed to its clipped objective function. In summary, the discussion seemed to reveal a trade-off between the speed of performance improvement and training stability, indicating that there is no single algorithm that is best on all aspects of performance, training speed, and stability.

Chapter 6

Conclusion

This chapter presents and discusses the main implications of the work done and the results achieved in this thesis, both in a narrow sense, regarding the research questions, and the broad picture regarding contributions to the field. Section 6.1 gives an overview of the entire thesis. Section 6.2 reviews the goal statement and aims to answer the research questions posed in this thesis. In section 6.3 the contributions of this thesis to the fields of wildlife management and reinforcement learning are presented. Section 6.4 briefly discusses the ethical implications of the RL system built in this thesis. Finally, section 6.5 presents suggestions for future work.

6.1 Overview

The background and motivation for this thesis were presented in chapter 1. A research goal and two research questions were also formulated in this chapter.

Chapter 2 introduced key concepts in wildlife management and reinforcement learning, which were necessary to understand the related work and methodology of this thesis. The first section of this chapter defined wildlife management and relevant actions in the field. Section 2.2 started by introducing a simple, but popular model for a predator-prey ecosystem. This was based on the Lotka-Volterra equations. Throughout this section, the model was developed and extended to become more realistic and thus more complex. The final model simulates a tri-trophic predator-prey ecosystem. This model is the basis for the RL envir-

onment in this thesis. The last section of chapter 2 serves as an introduction to reinforcement learning. Different approaches in this field, such as Q-learning and Actor-Critic, are presented. These approaches serve as the underlying theory for the DRL algorithms tested and compared in this thesis.

Chapter 3 presents related work both in the field of wildlife management and in the field of reinforcement learning. The first section describes how a structured literature review was carried out to find the related work. Traditional, non-AI approaches to wildlife management are presented in section 3.2. While these approaches serve as a good basis for modeling the RL environment in this thesis, their limitations are also discussed. The subsequent sections present work done in the field of reinforcement learning. This includes the application of RL to wildlife management and grid environments, but also a detailed description of the DRL algorithms tested in this thesis.

Chapter 4 describes the methodology for creating an RL environment and testing RL algorithms on it to answer the research questions of this thesis. This chapter presents an overview of the RL system built, as well as a detailed view of its different components. An in-depth explanation of how the RL environment, also referred to as the spatio-temporal wildlife environment, functions, is given. Implementation details regarding the DRL algorithms tested on this RL environment are also presented. Finally, an experimental plan is laid out, which outlines how the DRL algorithms are going to be tested on the RL environment to generate results that can answer the research questions.

The results of the experiments are presented in chapter 5. This includes comparisons of DRL algorithms with different action sets and also different DRL algorithms with the same action set. Interesting results regarding policy and biodiversity performance are also presented. This chapter also includes a more technical discussion of the results, with regard to the research questions. While a 2×2 action unit size enabled all DRL algorithms to perform their best, there were no clear best action multipliers. Overall, an action set of 2×2 action unit size and a 10x action multiplier seemed to perform well. The discussion also pointed to a trade-off between DRL algorithm performance over training time, and training stability.

6.2 Goal Evaluation

As stated in section 1.2, the goal of this thesis was to explore the use of DRL algorithms on the task of spatio-temporal wildlife management. To do this, two research goals were posed. These will now be answered and discussed. For convenience, the research questions will be repeated in this section.

Research Question 1

> *What action set yields the best performance on the task of spatio-temporal wildlife management, when accounting for the costs that it entails?*

In this thesis, the reward the RL agent received from the RL environment was used as a measure of its performance on the task of spatio-temporal wildlife management. The reward function also accounts for practical, real-life costs of executing different actions. The action set consists of two parameters, an action unit size, and an action multiplier. Based on initial testing of the RL environment, three candidate values for each of the parameters were chosen to be compared. The candidate values for action unit size were 2×2, 3×3, and 4×4. For the action multiplier, the candidate values were 5x, 10x, and 15x. The results show that all DRL algorithms receive the highest average episode reward with an action unit size of 2×2, compared to 3×3 and 4×4. However, for the action multiplier, there was no value that enabled the best performance for all DRL algorithms. The "best" action set is therefore dependent on which DRL algorithm is trained with it. This seems logical, especially considering that the RL agent employs different approaches to the task depending on which DRL algorithm it is trained with.

While there is no single best action multiplier, the 10x value stands out as consistently performing on average or better for all DRL algorithms. The action set consisting of a 2×2 action unit size and a 10x action multiplier, therefore, seems like a sensible choice.

It is important to highlight that not all combinations of candidate values for action unit size and action multiplier were tested, as it would be too resource-intensive and time-consuming. Therefore, one cannot with complete certainty determine that there are no better action sets.

It is also worth noting that the results collected in this thesis come from a particular tri-trophic predator-prey simulation with a specific set of parameters. Hence,

they are only valid for this specific situation. Wildlife management is very complex and can include fewer or more species, different species, or even different types of relationships between animals to name a few important factors. No two ecosystems are the same. While traditional RL environments, such as Atari 2600 games, have a defined action set, wildlife management does not. It is therefore worth investing time into finding, if not the best, a good action set for each situation.

In reality, one single action set is most likely not ideal for all different states of a wildlife environment. Ideally, the RL agent would be able to place different types of action units for different situations. However, this would considerably increase the complexity of the action space.

Research Question 2

> *Which DRL algorithm yields the best performance on the task of spatio-temporal wildlife management: Deep Q-Network (DQN), Advantage Actor-Critic (A2C), or Proximal Policy Optimization (PPO)? What are the trade-offs between these?*

Research question 2 was posed to compare and evaluate the performance of three popular DRL algorithms: DQN, A2C, and PPO. As different algorithms have different approaches to learning to navigate RL environments, an essential part of this question was to find out what trade-offs they had.

Since all algorithms performed reasonably well with the action set consisting of a 2×2 action unit size and a 10x action multiplier, their performances were compared when trained with this action set. The results revealed a trade-off between performance improvement over training time and training stability. A2C quickly, both in terms of environment steps and wall-clock time, improved its average episode reward. However, it seems unstable and shows great variability between its runs. PPO on the other hand slowly improved over time but was very stable. DQN seems to fall between A2C and PPO in both performance over training time and training stability.

It is worth noting that DQN is significantly more sample-efficient than A2C and PPO, and it runs sequentially, meaning that it only runs on one core. Therefore, given the hardware available for training, DQN might be a sensible choice. However, most modern computers have multiple CPU cores and therefore have the ability to run multiple RL environment instances in parallel during training. A2C and PPO make use of this ability and are therefore much faster than DQN when it comes to training over environment steps with regard to wall-clock time.

Given the same wall-clock training time, A2C still performs the best, but the performance gap is considerably smaller, as figure 5.9 shows. PPO seems to catch up to the other algorithms in terms of average episode reward, while also offering remarkable stability, much thanks to its clipped objective function. In contrast to A2C, it shows a dynamic strategy for the task of wildlife management and also scores the highest on the biodiversity metrics. One could therefore argue that, while there is a performance over training time and stability trade-off in the initial stages of training, PPO shows the most promise as training time increases.

6.3 Contributions

This thesis contributes to the fields of wildlife management and reinforcement learning in several ways. To my knowledge, RL has not been applied to the task of spatio-temporal *manipulative* wildlife management. For RL this seems to be a novel problem domain, and for manipulative wildlife management, RL appears to be a new solution technique. The work closest related to this thesis is the CAPTAIN project presented in section 3.4.4, which deals with conservation, a form of *custodial* wildlife management.

This thesis contributes to the mentioned fields by being the first attempt at applying DRL to the task of manipulative wildlife management. The thesis investigates the best action set for the RL agent and compares the performances of three popular DRL algorithms: DQN, A2C, and PPO on this task. The thesis also offers a technical discussion of the results obtained and a review of the strength and weaknesses of each algorithm in the context of this task.

Another contribution of this thesis comes in the form of a byproduct. To investigate the research questions posed in this thesis a spatio-temporal wildlife environment was created as the RL environment of this thesis. To my knowledge, this is the first RL spatio-temporal manipulative wildlife environment made. The environment was built using the Python library Gymnasium and follows its standard interface. Thus, it is very simple to train different algorithms from RL libraries compatible with Gymnasium on the RL environment created in this thesis. Researchers who would want to do further work in the fields of spatio-temporal manipulative wildlife management and reinforcement learning would not need to create their own RL environment if their RL algorithms are compatible with Gymnasium. The RL environment created in this thesis is publicly available at `https://github.com/AnmolS99/BioGym/`.

In the broader picture, this thesis highlights the potential of DRL algorithms to maintain biodiversity. All DRL algorithms, for nearly all action sets tested in this thesis, were able to improve their performance in the RL environment. This indicates that the RL agent learns something valuable when trained with the algorithms and that the species populations were more frequently above their critical threshold. Thus, this thesis has shown that DRL has the potential to be a valuable tool in the efforts to halt the ongoing biodiversity crisis and keep more sustainable ecosystems.

6.4 Ethical considerations

While the RL agents trained in this thesis are far from being ready for real-life use, it is worth briefly reflecting upon the ethical concerns that would exist if such a system was to ever be used in practice. RL, and a lot of AI techniques, are often referred to as "black boxes", as the internal process behind their decisions can be very complicated for a human to comprehend. For this thesis, it can be difficult to understand exactly why the RL agent trained with a DRL algorithm decides to take the actions it takes. The policy of an RL agent depends on factors such as reward function and learning algorithm, and understanding why an RL agent employs a certain policy can be very difficult. The lack of transparency in this process results in a lack of trust in the system. *Explainable AI* (XAI) has therefore become a larger focus in recent times and is highly relevant for the system built in this thesis as it could be used to affect real ecosystems with living animals. Taking suboptimal actions can have irreversible consequences for biodiversity. It is also worth noting that the RL agent does not consider important factors such as anthropogenic disturbance and animal welfare when taking actions. It is therefore important to remember that while RL can be a valuable tool, real-life decisions in this domain should be taken in accordance with experts in the field.

6.5 Future Work

As this thesis explores the relatively novel domain of wildlife management and DRL, there are many interesting directions future work can take. There are also some potential improvements to the system built in this thesis.

A rather simple yet resource-intensive improvement to the system built in this thesis would be to perform hyperparameter tuning. As discussed at the end

of section 5.2.2, hyperparameters have a significant effect on the performance of DRL algorithms. Given the hardware used for this thesis, it was deemed a too time-consuming exercise. However, if sufficient resources are available, hyperparameter tuning could improve the performances of the DRL algorithms trained in this thesis. There are multiple software frameworks for tuning DRL algorithm hyperparameters. *RL Baselines3 Zoo* is a training framework that uses SB3, the RL library used in this thesis. This framework uses *Optuna* to optimize hyperparameters (Akiba et al., 2019).

The RL environment built in this thesis could also benefit from an alternative approach to the representation of its state space. The state of the spatio-temporal wildlife simulation is composed of three grids, which are built up of cells. Naturally, an alternative representation would be to treat the grids as images, where the cells serve as pixels. This would not change the dynamics of the RL environment but opens up the possibility for *Convolutional Neural Networks* (CNNs) to be used to process the RL environment state in DRL algorithms. CNNs are widely used for image classification as they are efficient at extracting useful features from images. This could be useful for the state of the wildlife environment, especially with regard to the spatial distribution of species populations. For the small grid sizes used in this thesis, ANNs seem to be sufficient, however, for larger grids, CNNs might be a more suitable choice. It should be noted that the original DQN algorithm, presented by Mnih et al. (2013) to play Atari 2600 games, used CNNs and achieved great performance. This is also an approach that could be applied to the CAPTAIN project described in section 3.4.4, as it has a similar state representation as the one in this thesis.

When it comes to the RL environment, further research should be undertaken to explore and model various wildlife environments. The RL environment in this thesis models a complex, but idealistic version of a wildlife ecosystem. Ecosystems in real-life are significantly more complex and are influenced by a large number of factors such as climate, disease, habitat, and anthropogenic disturbance, to name a few. The study of the applicability of DRL algorithms to perform wildlife management in different, and more realistic ecosystems is therefore an important step in determining its potential and practical value in real-life wildlife management.

Further research might also explore the performance of other RL algorithms on the task of spatio-temporal manipulative wildlife management. The CAPTAIN project utilized evolutionary strategies (ES), as described in section 3.4.4. ES presents a different approach to learning than the DRL algorithms tested in this thesis. While DQN, A2C, and PPO update their neural networks by exploring states and computing gradients, ES perturbs the neural network weights, also

called policy, multiple times and measures each policy's performance through a fitness function. The neural network weights are then updated based on the fitness of the policies. As ES have a fundamentally different approach than traditional RL algorithms, they have their strength and weaknesses. ES avoid the computational cost of backpropagation and the risk of exploding/vanishing gradients, however, they seem to be less effective when there is a high correlation between action and reward (Salimans et al., 2017). Its successful application on a similar problem domain in the CAPTAIN project makes the application of ES on the spatio-temporal wildlife simulation worth investigating in the future.

This thesis has investigated the use of DRL algorithms for wildlife management. The results have shown that the RL agent is able to learn and navigate the spatio-temporal wildlife environment more efficiently as it is being trained with DRL algorithms. Thus, this thesis has laid down a solid groundwork for further research and work in the fields of wildlife management and reinforcement learning.

Bibliography

Akiba, Takuya et al. (July 2019). 'Optuna: A Next-generation Hyperparameter Optimization Framework'. en. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Anchorage AK USA: ACM, pp. 2623–2631. ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330701. URL: https://dl.acm.org/doi/10.1145/3292500.3330701 (visited on 6th June 2023).

Altamimi, Abdulelah et al. (2022). 'Large-Scale Wildfire Mitigation Through Deep Reinforcement Learning'. In: *Frontiers in Forests and Global Change* 5. ISSN: 2624-893X. URL: https://www.frontiersin.org/articles/10.3389/ffgc.2022.734330 (visited on 13th Dec. 2022).

Bhupendra, Pauli Virtanen (2017). *Matplotlib: lotka volterra tutorial — SciPy Cookbook documentation*. URL: https://scipy-cookbook.readthedocs.io/items/LoktaVolterraTutorial.html (visited on 3rd Nov. 2022).

Braylan, Alex et al. (2015). 'Frame Skip Is a Powerful Parameter for Learning to Play Atari'. en. In.

Britannica (2022). *carrying capacity — biology — Britannica*. en. URL: https://www.britannica.com/science/carrying-capacity (visited on 3rd Nov. 2022).

Fritz, Kyle (Aug. 2022). '5 Ways AI is Helping Wildlife Conservation'. en-US. In: *AI Time Journal - Artificial Intelligence, Automation, Work and Business*. Section: Healthcare. URL: https://www.aitimejournal.com/how-ai-is-helping-wildlife-conservation (visited on 27th Oct. 2022).

Fryxell, John M. (2014). *Wildlife ecology, conservation, and management*. eng. 3rd ed. Chichester: Wiley Blackwell. ISBN: 978-1-118-29107-8.

Fryxell, John M. et al. (2020). 'Anthropogenic Disturbance and Population Viability of Woodland Caribou in Ontario'. en. In: *The Journal of Wildlife Management* 84.4. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jwmg.21829, pp. 636–650. ISSN: 1937-2817. DOI: 10.1002/jwmg.21829. URL: https:

//onlinelibrary.wiley.com/doi/abs/10.1002/jwmg.21829 (visited on 11th Nov. 2022).

Gymnasium (June 2023). original-date: 2022-09-08T01:58:05Z. URL: https://github.com/Farama-Foundation/Gymnasium (visited on 9th June 2023).

Hastings, Alan and Thomas Powell (1991). 'Chaos in a Three-Species Food Chain'. In: *Ecology* 72.3. Publisher: Ecological Society of America, pp. 896–903. ISSN: 0012-9658. DOI: 10.2307/1940591. URL: https://www.jstor.org/stable/1940591 (visited on 6th Nov. 2022).

Hauge, Jens Gabriel (Mar. 2020). *enzymkinetikk*. no. URL: http://snl.no/enzymkinetikk (visited on 19th Jan. 2023).

Henderson, Peter et al. (Jan. 2019). *Deep Reinforcement Learning that Matters*. arXiv:1709.06560 [cs, stat]. URL: http://arxiv.org/abs/1709.06560 (visited on 8th May 2023).

Holling, C. S. (1959). 'The Components of Predation as Revealed by a Study of Small-Mammal Predation of the European Pine Sawfly'. In: *The Canadian Entomologist* 91.5, pp. 293–320. DOI: 10.4039/Ent91293-5.

Hornik, Kurt, Maxwell Stinchcombe and Halbert White (Jan. 1989). 'Multilayer feedforward networks are universal approximators'. en. In: *Neural Networks* 2.5, pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208 (visited on 1st Dec. 2022).

Lapeyrolerie, Marcus et al. (June 2021). *Deep Reinforcement Learning for Conservation Decisions*. arXiv:2106.08272 [cs, q-bio]. URL: http://arxiv.org/abs/2106.08272 (visited on 6th Dec. 2022).

Lobry, Claude and Tewfik Sari (Nov. 2015). 'Migrations in the Rosenzweig-MacArthur model and the "atto-fox" problem'. en. In: *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* Volume 20 - 2015 - Special... P. 1990. ISSN: 1638-5713. DOI: 10.46298/arima.1990. URL: http://arima.inria.fr/020/pdf/vol.20.pp.95-125.pdf (visited on 8th Dec. 2022).

Lotka, Alfred James (1925). *Elements of Physical Biology*. en. Google-Books-ID: lsPQAAAAMAAJ. Williams & Wilkins. ISBN: 978-0-598-81268-1.

McMahon, Clive R. et al. (2010). 'Spatially explicit spreadsheet modelling for optimising the efficiency of reducing invasive animal density'. en. In: *Methods in Ecology and Evolution* 1.1. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.2041-210X.2009.00002.x, pp. 53–68. ISSN: 2041-210X. DOI: 10.1111/j.2041-210X.2009.00002.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2041-210X.2009.00002.x (visited on 14th Nov. 2022).

Mengak, Michael T. (2008). *Wildlife Management*. URL: https://sites.google.com/site/forestryencyclopedia/Home/Wildlife%20Management (visited on 26th Oct. 2022).

Mnih, Volodymyr et al. (Dec. 2013). *Playing Atari with Deep Reinforcement Learning*. arXiv:1312.5602 [cs]. URL: http://arxiv.org/abs/1312.5602 (visited on 23rd Mar. 2023).

Mnih, Volodymyr et al. (Feb. 2015). 'Human-level control through deep reinforcement learning'. en. In: *Nature* 518.7540. Number: 7540 Publisher: Nature Publishing Group, pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236 (visited on 28th Mar. 2023).

Mnih, Volodymyr et al. (June 2016). *Asynchronous Methods for Deep Reinforcement Learning*. arXiv:1602.01783 [cs]. URL: http://arxiv.org/abs/1602.01783 (visited on 17th Apr. 2023).

Nagy-Reis, Mariana et al. (2020). ''WildLift': An Open-Source Tool to Guide Decisions for Wildlife Conservation'. In: *Frontiers in Ecology and Evolution* 8. ISSN: 2296-701X. URL: https://www.frontiersin.org/articles/10.3389/fevo.2020.564508 (visited on 11th Nov. 2022).

Raffin, Antonin et al. (Nov. 2021). 'Stable-Baselines3: Reliable Reinforcement Learning Implementations'. en. In.

Riedmiller, Martin (2005). 'Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method'. en. In: *Machine Learning: ECML 2005*. Ed. by David Hutchison et al. Vol. 3720. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 317–328. ISBN: 978-3-540-29243-2 978-3-540-31692-3. DOI: 10.1007/11564096_32. URL: http://link.springer.com/10.1007/11564096_32 (visited on 25th Mar. 2023).

Rosenzweig, M. L. and R. H. MacArthur (1963). 'Graphical Representation and Stability Conditions of Predator-Prey Interactions'. In: *The American Naturalist* 97.895, pp. 209–223. URL: http://www.jstor.org/stable/2458702.

Russell, Stuart J. (2016). *Artificial intelligence: a modern approach*. eng. 3rd ed.; Global ed. Prentice Hall series in artificial intelligence. Boston, Mass.: Pearson. ISBN: 978-1-292-15397-1.

Salimans, Tim et al. (Sept. 2017). *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. arXiv:1703.03864 [cs, stat]. URL: http://arxiv.org/abs/1703.03864 (visited on 5th Dec. 2022).

Schulman, John et al. (Aug. 2017a). *Proximal Policy Optimization Algorithms*. arXiv:1707.06347 [cs]. URL: http://arxiv.org/abs/1707.06347 (visited on 21st Apr. 2023).

Schulman, John et al. (Apr. 2017b). *Trust Region Policy Optimization*. arXiv:1502.05477 [cs]. URL: http://arxiv.org/abs/1502.05477 (visited on 24th Apr. 2023).

Shannon, C E (1948). 'A Mathematical Theory of Communication'. en. In.

Silvestro, Daniele et al. (May 2022). 'Improving biodiversity protection through artificial intelligence'. en. In: *Nature Sustainability* 5.5. Number: 5 Publisher:

Nature Publishing Group, pp. 415–424. ISSN: 2398-9629. DOI: 10.1038/s41893-022-00851-6. URL: https://www.nature.com/articles/s41893-022-00851-6 (visited on 1st Dec. 2022).

Singh, Anmol (Dec. 2022). *Reinforcement Learning for Spatio-Temporal Wildlife Management*.

Spellerberg, Ian and Peter Fedor (May 2003). 'A tribute to Claude Shannon (1916–2001) and a plea for more rigorous use of species richness, species diversity and the 'Shannon–Wiener' Index'. In: *Global Ecology & Biogeography* 12, pp. 177–179. DOI: 10.1046/j.1466-822X.2003.00015.x.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement learning: An Introduction*. eng. Adaptive computation and machine learning. The MIT Press. ISBN: 978-0-262-03924-6.

Tanner, James T. (1975). 'The Stability and the Intrinsic Growth Rates of Prey and Predator Populations'. In: *Ecology* 56.4. Publisher: Ecological Society of America, pp. 855–867. ISSN: 0012-9658. DOI: 10.2307/1936296. URL: https://www.jstor.org/stable/1936296 (visited on 5th Nov. 2022).

Volterra, Vito (Apr. 1928). 'Variations and Fluctuations of the Number of Individuals in Animal Species living together'. In: *ICES Journal of Marine Science* 3.1, pp. 3–51. ISSN: 1054-3139. DOI: 10.1093/icesjms/3.1.3. URL: https://doi.org/10.1093/icesjms/3.1.3 (visited on 28th Oct. 2022).

Wang, Leye et al. (May 2018). *Cell Selection with Deep Reinforcement Learning in Sparse Mobile Crowdsensing*. arXiv:1804.07047 [cs]. URL: http://arxiv.org/abs/1804.07047 (visited on 24th Nov. 2022).

Wiggins, Natasha et al. (Aug. 2014). 'Using Spatio-Temporal modelling as a decision support tool for management of a native pest herbivore'. In: *Applied Ecology and Environmental Reseach* 12. DOI: 10.15666/aeer/1201_163178.

Wilson, Kerrie A. et al. (Mar. 2006). 'Prioritizing global conservation efforts'. en. In: *Nature* 440.7082. Number: 7082 Publisher: Nature Publishing Group, pp. 337–340. ISSN: 1476-4687. DOI: 10.1038/nature04366. URL: https://www.nature.com/articles/nature04366 (visited on 10th Nov. 2022).

Wolpert, David (Mar. 1996). 'The Lack of A Priori Distinctions Between Learning Algorithms'. In: *Neural Computation* 8. DOI: 10.1162/neco.1996.8.7.1341.

Wu, Yuhuai et al. (Aug. 2017). *OpenAI Baselines: ACKTR & A2C*. en-US. URL: https://openai.com/research/openai-baselines-acktr-a2c (visited on 20th Apr. 2023).

WWF (2022a). *Living Planet Report 2022 – Building a nature-positive society*. URL: https://wwflpr.awsassets.panda.org/downloads/lpr_2022_full_report.pdf (visited on 24th Oct. 2022).

— (2022b). *Wildlife Conservation – Conserve threatened wildlife and wild places to sustain life on Earth*. URL: https://www.worldwildlife.org/initiatives/wildlife-conservation (visited on 26th Oct. 2022).

Appendix A

DRL Algorithm Policies

Figures A.1, A.2 and A.3 display the policy (actions taken) of an RL agent over three consecutive time steps in a 9×9 spatio-temporal wildlife environment when trained on the different DRL algorithms.

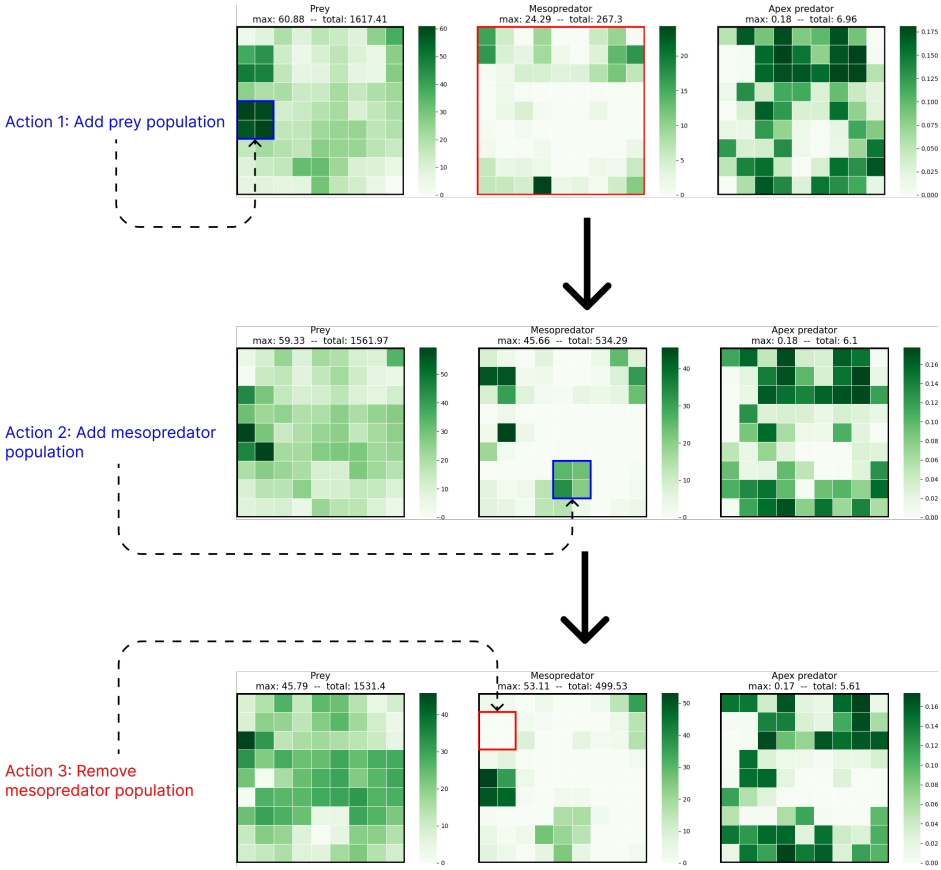


Figure A.1: A consecutive sequence of environment states and actions taken by an RL agent trained with the DQN algorithm. The best of 20 training runs over 200 000 time steps were used. The action set consists of an action unit size of 2×2 and an action multiplier of 10x. Each row shows the population distribution of the three species at a certain time. The red outline on the grid signals that the total species population is below its critical threshold.

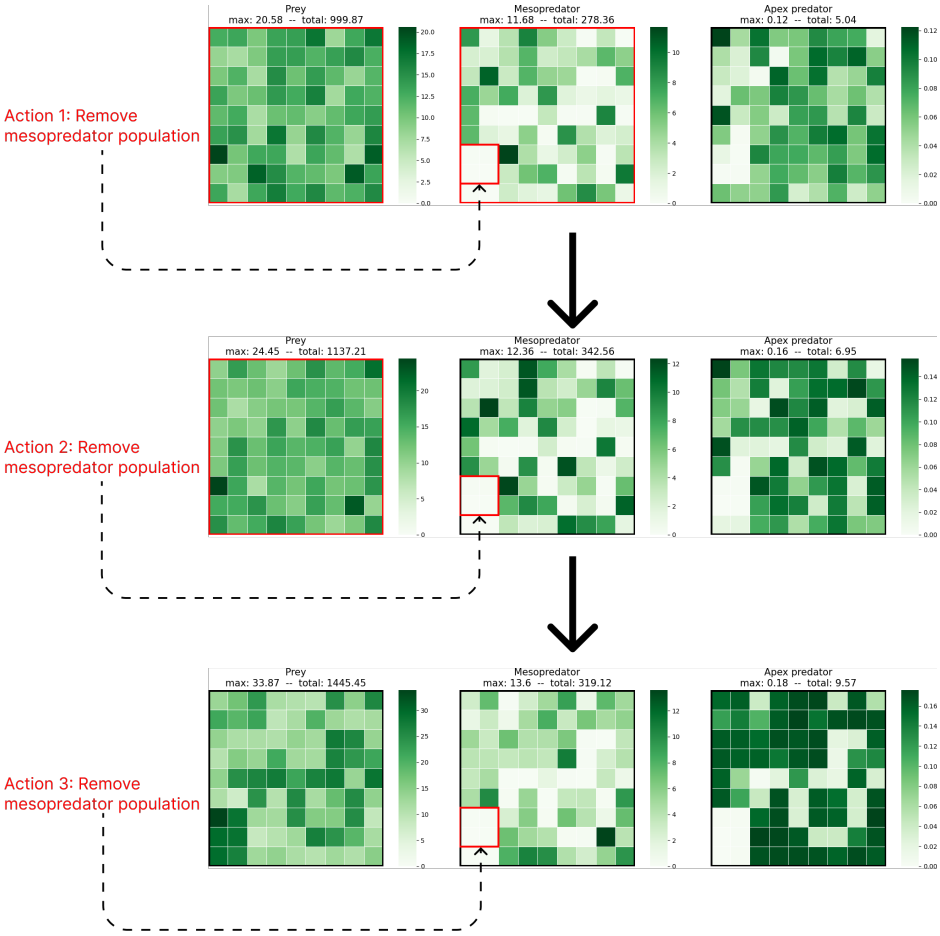


Figure A.2: A consecutive sequence of environment states and actions taken by an RL agent trained with the A2C algorithm. The best of 20 training runs over 800 000 time steps were used. The action set consists of an action unit size of 2×2 and an action multiplier of 10x. Each row shows the population distribution of the three species at a certain time. Red outline on grids signals that the total species population is below its critical threshold.

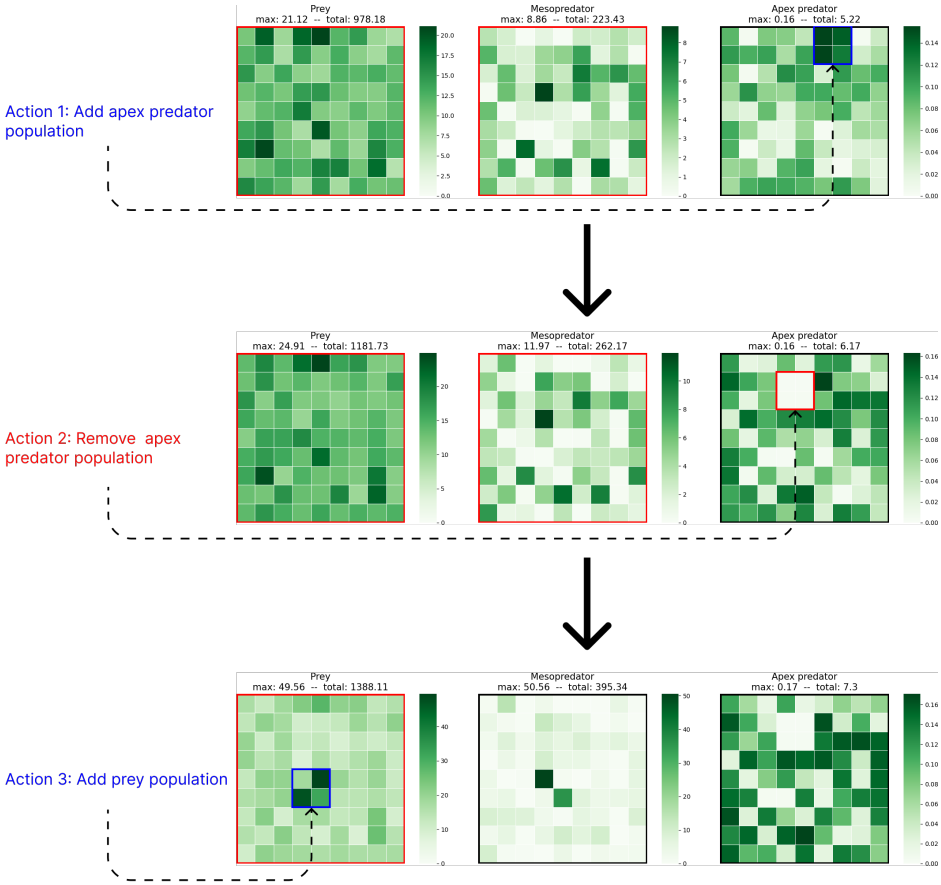
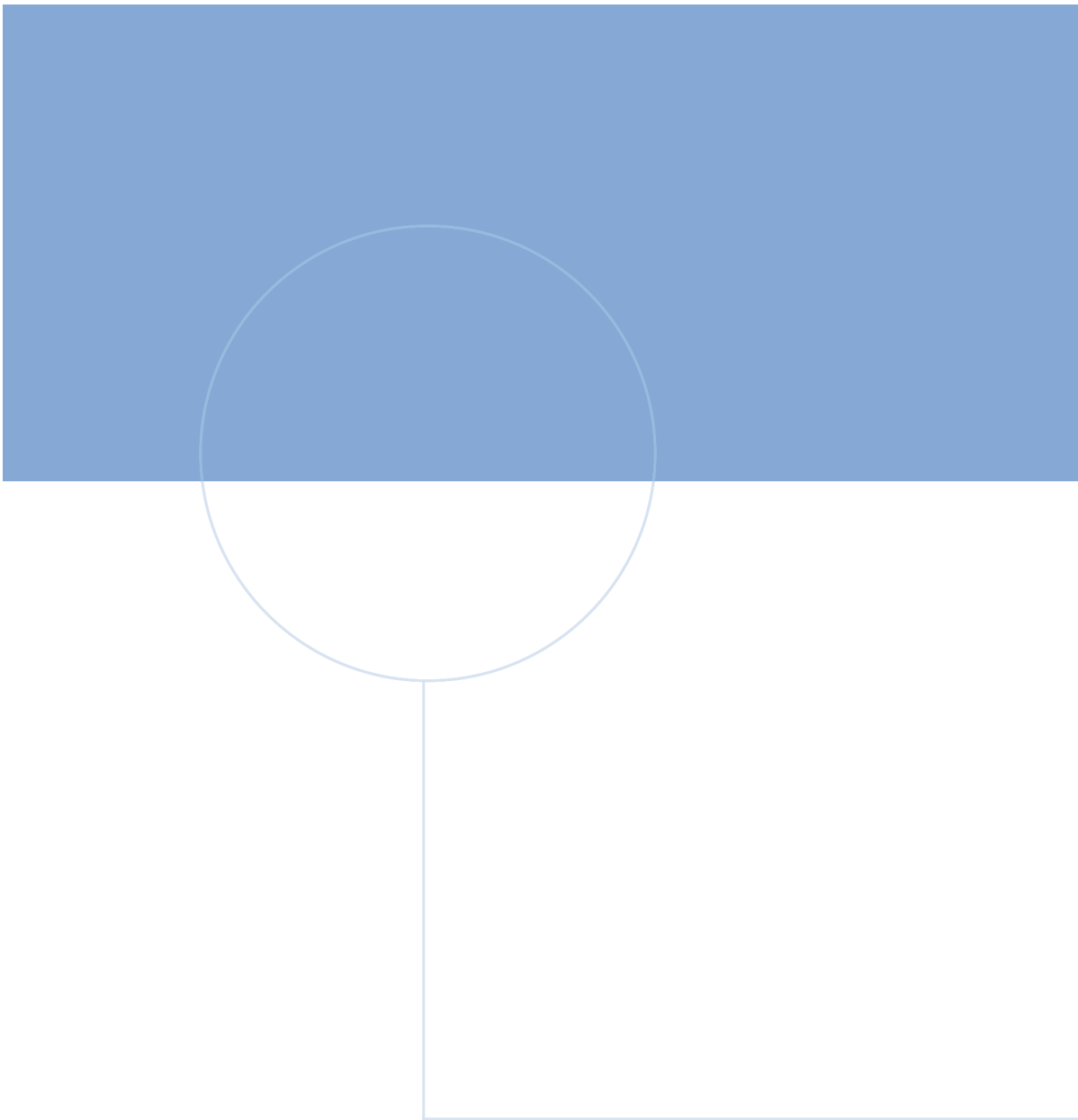


Figure A.3: A consecutive sequence of environment states and actions taken by an RL agent trained with the PPO algorithm. The best of 20 training runs over 802 816 time steps were used. The action set consists of an action unit size of 2×2 and an action multiplier of 10x. Each row shows the population distribution of the three species at a certain time. Red outline on grids signals that the total species population is below its critical threshold.



NTNU

Norwegian University of
Science and Technology