INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE INTELIGÊNCIA ARTIFICIAL




Criar Animais Virtuais Através de Machine Learning

HUGO TIAGO TRINDADE ROCHA
Outubro de 2022

Creating Virtual Animals Through Machine Learning

Hugo Tiago Trindade Rocha

Student No.: 1200131

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Artificial Intelligence

Supervisor: Doutora Isabel Cecília Correia da Silva Praça Gomes Pereira, Professora Coordenadora do Instituto Superior de Engenharia do Instituto Politécnico do Porto

Evaluation Committee:
President:
Doutora Ana Maria Neves Almeida Baptista Figueiredo, Professora Coordenadora do Instituto Superior de Engenharia do Instituto Politécnico do Porto

Members:
Doutor Carlos Fernando da Silva Ramos, Professor Coordenador Principal do Instituto Superior de Engenharia do Instituto Politécnico do Porto
Doutora Isabel Cecília Correia da Silva Praça Gomes Pereira, Professora Coordenadora do Instituto Superior de Engenharia do Instituto Politécnico do Porto

Porto, October 25, 2022

Abstract

Approximately 42 percent of threatened or endangered species are at risk due to invasive species. Some invasive species find the new habitat by themselves during migrations, others are misplaced by humans, be it by mistake or necessity.

This project aims to create a virtual habitat, populated by intelligent agents that represent the animals present in it. Programmers and scientists can add invasive species, and simulate what might happen. This will allow a more proactive response to this type of crisis.

Different data-driven models are explored in order to find the best one for the problem at hands.

Game engines are discussed, they have improved greatly over the last decade, and are accessible to everyone. Reliable tools to build simple or complex prototypes that give us graphic representations that can be photo realistic.

Keywords: Artificial Intelligence, Intelligent Agents, Artificial Neural Network, ANN, Fuzzy Logic, FL, Adaptive Neuro Fuzzy Inference System, ANFIS, Unity, Habitat Simulation, ML-Agents.

Resumo

Aproximadamente 42 por cento das espécies em vias de extinção estão em risco devido a espécies invasoras. Algumas dessas espécies invasoras chegam aos novos habitats através de migrações, outras chegam através da mão humana, voluntaria ou involuntariamente.

Este projeto tem como objetivo criar um habitat virtual, com agentes inteligentes que representam os animais presentes nesse mesmo habitat. Programadores e cientistas poderão adicionar espécies invasoras, e simular o que pode acontecer. Isto irá permitir uma resposta mais proativa quando estes tipos de crises acontecem.

Diferentes modelos orientados a dados são explorados, a fim de perceber qual será o melhor para resolver o problema.

Game engines são discutidos, este tipo de ferramenta tem evoluído bastante ao longo da última década, são ferramentas grátis, que podem ser usadas para criar protótipos com gráficos simples, ou foto realistas.

Palavras-chave: Artificial Intelligence, Intelligent Agents, Artificial Neural Network, ANN, Fuzzy Logic, FL, Adaptive Neuro Fuzzy Inference System, ANFIS, Unity, Habitat Simulation, ML-Agents.

Contents

List of Figures

List of Source Code

List of Acronyms

AI	Artificial Intelligence.
ANFIS	Adaptive Neuro Fuzzy Inference System.
ANN	Artificial Neural Network.
CNN	Convolutional Neural Network.
FL	Fuzzy Logic.
FNN	Feedforward Neural Network.
GAIL	Generative Adversarial Imitation Learning.
IL	Imitation Learning.
KSONN	Kohonen Self Organizing Neural Network.
ML	Machine Learning.
MNN	Modular Neural Network.
PPO	Proximal Policy Optimization.
RBFNN	Radial Basis Function Neural Network.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
SAC	Soft Actor Critic.
TSK	Takagi-Sugeno-Kang.

Chapter 1

Introduction

This chapter contains a brief presentation of the project, a description about the context and the problem that it aims solve.

1.1 Contextualization

Since there's life in our planet, every single species live in a daily struggle for survival. The world is in a constant state of stress due to its fragile equilibrium. The famous "survival of the fittest" (Spencer 1864), is a key process "On the Origin of Species" (Darwin 1859), nature puts itself to the test constantly, hindering those who don't have the capacity to adapt, and favoring those who can, with a place in the ecosystem.

A species capacity to adapt can be put to the test when a invasive species arrives in a new habitat. This event can trigger a brutal clash between the indigenous species and the invasive one, by the end of it, the new ecosystem will eliminate those who didn't adapt and adjust itself accordingly. This is a natural process, part of the evolution, and unless the outcome is catastrophic for the environment, humans should not interfere if possible.

When talking about invasive species introduced by humans, everyone agrees that we have the moral and vital necessity of interfere, and try to repair the damage done at the best of our capacity. We might be part of the ecosystem like all the other animals, but we have shaped the land too drastically due to our intellect and technology, so those same attributes should be used to mend the world we live in.

The destruction of the ecosystem, is another factor for the extinction of many species that lose the habitat they evolved for.

1.2 Problem Description

Approximately 42 percent of threatened or endangered species are at risk due to invasive species (Federation 2020). Through past and recent history, we have many examples of humans introducing invasive species, for example feral pigs who will eat almost anything and spread diseases (Federation 2020). Humans over hunting can also cause species to be endangered or extinct, like for example the Tasmanian Tiger (Peace 2020).

To fix the problems stated above, many efforts have been done in order to reintroduce species into the wild, some are still present in the habitat, but their numbers are dwelling, others where completely drove away.

This solution is problematic and can be unpredictable, adding species into a habitat requires some variables to be present, like for example food and shelter. There is more to take into account, the habitat could have been invaded by new species, who is itself endangered. Nature is highly complex, and extremely hard to predict.

1.3 Objectives

This project aims to simulate ecosystems, with intelligent agents that represent different individuals from different species, and try to simulate what happens when a given species is added/removed from the ecosystem. This simulations can help experts predict the outcomes of such complex scenarios.

The more complex the system, the more accurate the predictions will be, however there are many variables to take into account, and the smallest factor can have a butterfly effect.

For the sake of simplicity, the simulation will be performed with fictitious animals inspired by real ones. Every single aspect of the agents will be parameterized, since flexibility is important to give experts the ability to create realistic animals.

1.4 Research Hypotheses

Since the objective of this project is to simulate a natural habitat in a virtual environment, first of all the intelligent agents need to learn the behavior of the animals that live in the habitat. This can be hard to do the classic way where a programmer would have to code each and every single interaction and place multiple systems to make sure everything works as planned. Reinforcement Learning (RL) with Imitation Learning (IL) could be the answer to help developers and experts in the task of creating virtual replicas of the animals.

To achieve this objective there are two questions that need to be answered first, that will help understand if this solution is possible or not.

1. Can experts create realistic behaviors by using RL with IL?

   Can IL help experts and developers recreate a realistic behavior. An expert can control the intelligent agent in the virtual environment, and show the agent whats expected from it. The model would be saved when the performance was satisfactory, and used in the simulation.

2. Can intelligent agents interactions be used to predict changes in a given habitat population?

   If this simulation can predict changes in the food available and number of individuals of each species that are left.

1.5 Document Structure

This dissertation has a structure composed by 4 chapters, they aim to explain each part of the process in a simple and comprehensive way.

It begins with the introductory chapter which explains the motivation behind the study, the problem it aims to explore and solve. The hypotheses and objectives are laid out in order to understand the plan that was followed.

The second chapter discusses the state of art, where the available technology's are explored to give a broad picture of which technology's are available to solve the problem. There's also mentions to papers that used the same tools that will be used in this study, or try to solve the same problems.

The third chapter explains all the methods used, starts by showcasing the hardware used, the game engine chosen, the plugin used to create the artificial intelligence environment, and the algorithms used to train the intelligent agents. The intelligent agents development is also discussed, to explain how they were built and how they perceive the environment. The training results are showed and a simulation is with the developed intelligent agents is performed.

The forth chapter talks about the contributions of this study and discusses if the hypotheses have been proven and highlights future works that could be done.

Chapter 2

State of Art

In this chapter there will be examples of systems that try to simulate habitats, be it for predicting the outcome of changes, or just for entertainment, the models they used and some of the tools that were/can be used to build the virtual environments.

2.1 Fuzzy Logic

The term Fuzzy Logic Fuzzy Logic (FL) was created in 1965 on a proposal of fuzzy set theory by scientist Lotfi Zadeh (Hajek 2010). But its creation dates back to 1920s, as infinite-valued logic—notably by Łukasiewicz and Tarski (Pelletier 2000).

FL is based on degrees of truth rather than the usual true or false Boolean logic on which modern computers are based. This makes FL perfect to imitate living beings reasoning and cognition. Instead following a strict binary cases of truth, FL uses 0 and 1 as extreme cases of truth, but with various intermediate levels of truth. When talking about fuzzy models we can describe them as a mathematical way of representing vague or imprecise information. These models can recognize, represent, manipulate, interpret and use data/information that is vague and uncertain (Babuška 1998).

As it was highlighted previously, FL is used in projects where the certainty or lack of it, isn't clear. Such projects are found in the automobile systems, aerospace, natural language processing, medicine, among many others.

There are two main fuzzy systems, Mamdani and Takagi-Sugeno-Kang Takagi-Sugeno-Kang (TSK). Mamdani is one of the best known fuzzy systems, are designed to incorporate expert knowledge in form of IF-THEN rules expressed in natural language. This is an attractive feature for modelling and simulating social and other complex systems (Segismundo S. Izquierdo 2018). TSK model on the other hand is characterized by a high accuracy of modeling combined with a very fast learning process. The model was proposed as a systematic approach to generate fuzzy rules from a given set of input-output data and where the structure of the system if not known in advance (Miodrag Petkovic 2018).

2.2 Artificial Neural Network

Artificial Neural Networks Artificial Neural Network (ANN) are models that imitate the biological neural network. ANN were invented in 1943 by neurophysiologist Warren McCulloch and the logician Walter Pits (Kleene 1956).

In its simplest form, biological brains are a huge collection of neurons. Each neuron takes electrical or chemical signals as inputs through its many dendrites and transmits the output signal through its axon. Axons then make contact with other neurons at specialized junctions called synapses where they pass on they're output signals to other neurons, and this process repeats over millions and millions of times (Klimasauskas 1989).

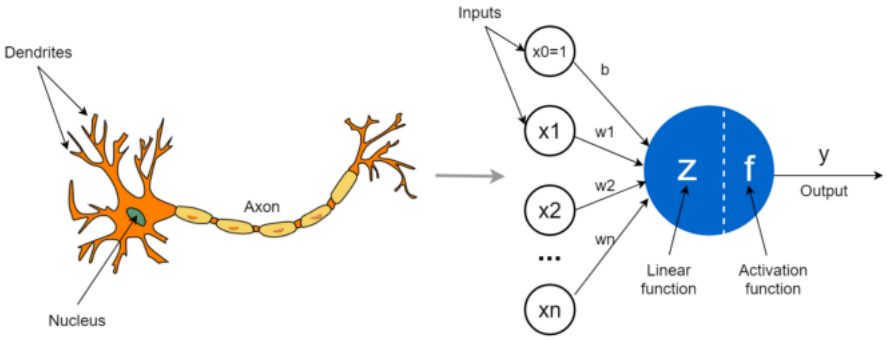

Figure 2.1: Biological neuron next to a artificial neuron (Pramoditha 2021).

By taking inspiration from the brain, ANN are collections of connected units named neurons, figure 2.1 shows a biological neuron next to its digital counterpart. The connection between the neurons can carry signals between them. Each connection carries a real number of value which determines the weight/strength of the signal. The units between the input units and the output units, are called hidden layer (N. Murata and Amari 1993). In figure 2.2 it's possible to see a basic representation of the architecture of a artificial neural network, containing the input and output layer, and between them two hidden layers, in the connections between the neurons the weights are also visible.

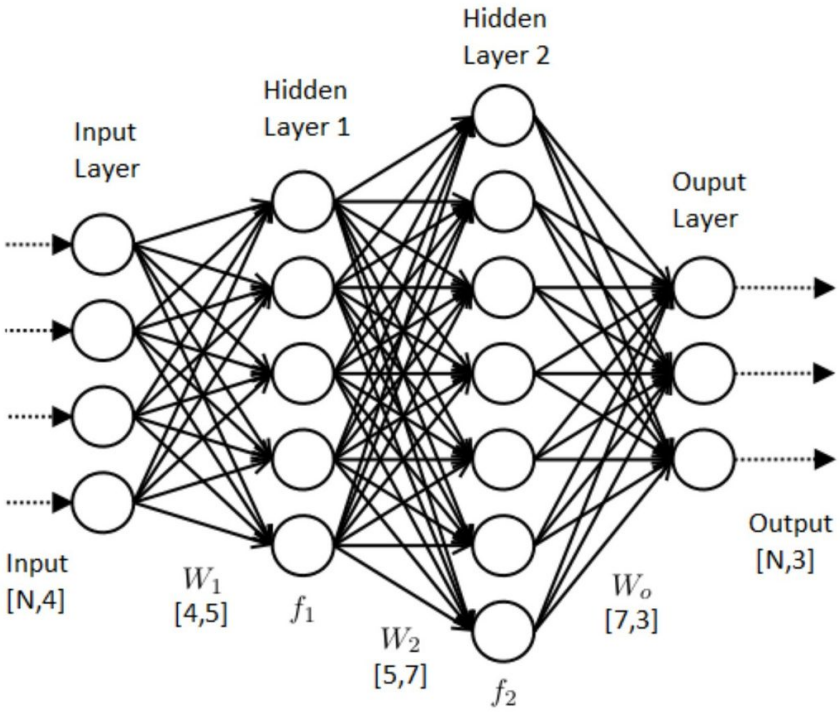

Figure 2.2: Artificial Neural Network architecture (Ahire 2018).

ANN instead of using the classic digital model of zeros and ones, it works by creating connections between processing elements called neurons like the ones in biological brains. The organization and weights of the connections determine the output (Laboratory 2021).

ANN are effective for predicting events when the networks have a large database of prior examples to work on. ANN are designed to recognize patterns. They interpret sensory data through some sort of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vector, into which all real-world data, such as images, sound, text or time series, must be translated (Mohaiminul Islam 2019).

ANN come in many different types, there are Feedforward Neural Network Feedforward Neural Network (FNN), Radial Basis Function Neural Network Radial Basis Function Neural Network (RBFNN), Kohonen Self Organizing Neural Network Kohonen Self Organizing Neural Network (KSONN), Recurrent Neural Network Recurrent Neural Network (RNN), Convolutional Neural Network Convolutional Neural Network (CNN) and Modular Neural Network Modular Neural Network (MNN). These types are implemented based on the mathematical operations and a set of parameters required to determine the output (J. A. Bradshaw 1991).

2.2.1 Feedforward Neural Network

FNN is one of the simplest form of ANN. Data or input travels in one direction. The data passes through the input nodes and exit on the output nodes. This ANN may or may not have the hidden layers. It has a front propagated wave and no back propagation by using classifying activation function usually. FNN can be applied to computer vision and speech recognition where classifying the target classes are complicated. FNN are responsive to noisy data and easy to maintain (Mohaiminul Islam 2019).

2.2.2 Radial Basis Function Neural Network

RBFNN consider the distance of a point with respect to the center. Its function have two layers, first where the features are combined with the Radial Basis Function in the inner layer and then the output of these features are taken into consideration while computing the same output in the next time-step which is basically a memory (Mohaiminul Islam 2019). RBFNN can be applied to power restoration systems, those systems have increased in size and complexity, and for that reason the risk of a major power outage has increased as well. After blackouts power needs to be restored as soon as possible (Murphy 2012).

2.2.3 Kohonen Self Organizing Neural Network

KSONN map input vectors of arbitrary dimension to discrete map comprised of neurons. The map needs to be trained to create its own organization of the training data, it comprises of either one or two dimensions. When training the map the location of the neuron remains constant but the weights differ depending on the value (Mohaiminul Islam 2019). Since KSONN are good recognizing patterns in data, they are applied in medical analysis to cluster data into different categories. KSONN was able to classify patients having glomerular or tubular with an high accuracy (D. H. Hubel 1959).

2.2.4 Recurrent Neural Network

RNN work on the principle of saving the output of a layer and feeding this back to the input to help in predicting the outcome of the layer. The first layer is formed similar to the FNN with the product of the sum of the weights and the features. RNN process starts once this is computed, which means that from one time step to the next each neuron will remember some information it had in the previous time-step. Each neuron act like a memory cell in performing computations. During this process the RNN works on the front propagation and remember what information it needs for later use. If the prediction is wrong the learning rate or error correction is used to make small changes so that it will gradually work towards making the right prediction during the back propagation (Mohaiminul Islam 2019).

2.2.5 Convolutional Neural Network

CNN are similar to FNN, the neurons have learn-able weights and biases. Can be applied to signal and image processing which takes over OpenCV in field of computer vision (Mohaiminul Islam 2019). They are very accurate in images classification, so have been used in image analysis and recognition. This led them to be used on analyses of agriculture and weather features extracted from the open source satellites LSAT to predict the future growth and the yield of a particular land (Lawrence 1994).

2.2.6 Modular Neural Network

MNN have a collection of different networks working independently and contributing towards the output. Each ANN has a set of inputs which are unique compared to other networks constructing and performing sub-tasks. There is no interaction between the ANN in accomplishing the tasks. They breakdown a large computational process into smaller components decreasing its complexity (Mohaiminul Islam 2019).

2.3 Adaptive Neuro Fuzzy Inference System

Adaptive Neuro Fuzzy Inference System or adaptive network-based fuzzy inference system Adaptive Neuro Fuzzy Inference System (ANFIS) were developed in 1990s and are based on TSK fuzzy inference system (Jang 1991).

This model uses both models discussed previously, FL and ANN, so it captures the best of both, in a single model. The inference system corresponds to a set of fuzzy IF-THEN rules that have a learning capacity similar to nonlinear functions (Abraham 2005). ANFIS is considered a universal estimator (Jyh-Shing Roger Jang 1997).

ANFIS has a five-layer architecture, namely, fuzzy layer, product layer, normalized layer, defuzzifier layer and output layer. Output variables at each layer are calculated with input variables from the previous layer and parameters at each node (Sung-Uk Choi 2018).

2.4 Machine Learning

A way to introduce the machine learning Machine Learning (ML) methodology is by comparing it with the conventional engineering design flow. The conventional design flow starts with in-depth analysis of the problem domain, and then the definition of a mathematical model. The mathematical model captures the key features of the problem under study, and usually results of the work of many experts. The mathematical model is then leveraged to derive hand-crafted solutions to the problem that offer given optimally guarantees (Simeone 2018a).

ML on the other hand, collects large data sets, e.g., of labelled speech, images or videos, and uses this information to train general-purpose learning machines to carry out the desired task. It lets large amounts of data dictate algorithms and solutions. Instead of requiring a precise model of the set-up under study, ML requires the specification of an objective, of a generic model to be trained, and of an optimization technique (Simeone 2018a).

The following rules can help identify tasks for which ML may be useful (Erik Brynjolfsson 2017):

1. the task involves a function that maps well-defined inputs to well-defined outputs;
2. large data sets exist or can be created containing input-output pairs;
3. the task provides clear feedback with clearly definable goals and metrics;
4. the task does not involve long chains of logic or reasoning that depend on diverse background knowledge or common sense;
5. the task does not require detailed explanations for how the decision was made;
6. the task has a tolerance for error and no need for provably correct or optimal solutions;
7. the phenomenon or function being learned should not change rapidly over time; and
8. no specialized dexterity, physical skills, or mobility is required.

2.4.1 Supervised Learning

Supervised learning builds a knowledge base from patterns previously classified that supports and classifies new patterns. Its major task is to map the input features to an output

called class. The outcome is used to construct a model by examining the input patterns. The model can be used to correctly classify unseen instances. There are many supervised learning algorithms, some of which are Decision Trees, Random Forest, k-Nearest Neighbor, Logistic Regression, ANN, Support Vector Machines, Naive Base, Bayesian Networks (Y. C. A. Padmanabha Reddy 2018).

2.4.2 Unsupervised Learning

Unlike supervised learning, unsupervised learning tasks operate over unlabelled data sets consisting solely of inputs, and the general goal is that of discovering properties of the data (Simeone 2018b). Unsupervised learning studies how systems can learn to represent particular input patterns in a way that reflects the statistical structure of the overall collection of input patterns. Unlike supervised learning or RL, there are no explicit target outputs or environmental evaluations associated with each input. Instead it brings to bear prior biases as to what aspects of the structure of the input should be captured in the output. Examples of unsupervised learning algorithms are, Clustering, Expectation Maximization, Principal Component Analysis, Independent Component Analysis, Singular Value Decomposition (Y. C. A. Padmanabha Reddy 2018).

2.4.3 Reinforcement Learning

The concept of RL begins from the way the machine learns i.e. unsupervised learning technique. In unsupervised learning the machine has unlabeled data, and from it draws its own conclusions. RL can be used to teach a machine to perform specific task, like for example playing chess, chess has rules that the machine would have to follow. In this example the machine would be the agent, and the chess board would be the virtual environment. The agent performs actions in the virtual environment. In real life we are the agents that plays the game, according to what we need to do or what our brain tells us to do, and the physical chess board is the environment. But it is different for machines, because we know that we have to accomplish a task to win the game but the machine doesn't know that. For this purpose exists the reward and punishment approach. If the action did by the agent takes it towards the goal it is rewarded and if it takes it far from the goal it will be punished. In machine terms they are just some positive and negative values, positive means a reward and negative means it was punished. Figure 2.3 shows the RL architecture that helps visualise the process described previously. Examples of RL algorithms are Bellman Equation, Markov Decision Processes, Q-Learning, Deep Q-Learning and Proximal Policy Optimization Proximal Policy Optimization (PPO) (Mahajan 2020).

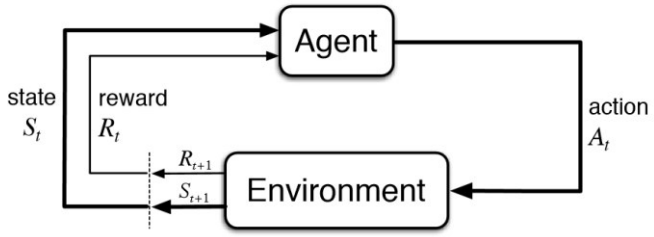

Figure 2.3: RL architecture (Bhatt 2018).

Proximal Policy Optimization

PPO is motivated by the question: how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse? PPO is a family of first-order methods that use a few other tricks to keep new policies close to old. PPO methods are simpler to implement, and seem to perform well. This is an on-policy algorithm, can be used for environments with either discrete or continuous action spaces and the implementation supports parallelization (John Schulman 2017).

Soft Actor Critic

Soft Actor Critic (SAC) is off-policy so it can learn from experiences collected at any time during the past. Collected experiences are placed in an experience replay buffer and randomly drawn during training. Which means SAC is significantly more sample-efficient, usually requiring 5-10 times less samples to learn the same task as PPO, on the down side, it will require more model updates. SAC can be a good choice for heavier or slower environments (ML-Agents 2021a).

2.4.4 Imitation Learning

In IL, the agent learns manipulation by observing an expert's demonstration. This process extracts information of the behavior and surrounding environment, while learning the mapping between the observation and the performance. The task of agent manipulation can be viewed as a Markov decision process, it encodes the action sequence of the expert into state-action pairs that are consistent with the expert. When the trained data available has good quality, the learning is very efficient since it is learning from trained data, and not from scratch (Bojarski M. 2016). In combination with RL, the speed and accuracy of IL can be improved, currently the methods of IL can be divided into behavior cloning, inverse RL and Generative Adversarial Imitation Learning (GAIL) (Kumar V. 2016). Figure 2.4 contains the classification of IL.

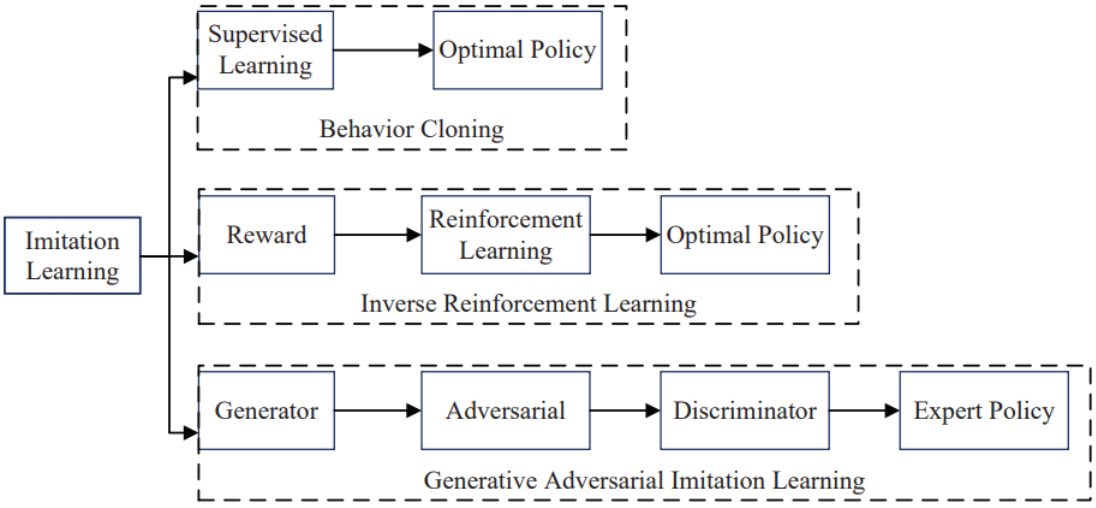

Figure 2.4: Classification of IL (Jiang Hua 2021).

Generative Adversarial Imitation Learning

In this study it was used GAIL, it is the type of IL available in ML-Agents. The method compares the difference between the generated strategy and the expert strategy. Iterative confrontation training can be performed to make the distribution between the expert and the agent as close as possible (Kuefler A. 2017). Pure learning methods and simple rewards functions, often result in non-natural behavior, and too rigid movement behaviors, algorithms like GAIL can generate more natural motion patterns from limited demonstrations without access to actions, it constructs strategies that can be reused to solve tasks when controlled by a higher-level controller (Merel J. 2017).

2.5 Game Engines

This section discusses two of the current best and most widely used game engines. Game engines are perfect for prototyping and developing games in a short period of time. When using such tools developers don't need to create everything from scratch, many functions common to every game like for example graphics rendering, input handlers, sound managers, objects collisions, gravity simulation and many more features, are already present, and easy to use without programming a single line of code. If the developers need any of those features more personalized to their project, they can create their own solutions as well.

Unreal Engine is a game engine developed by Tim Sweeney, founder of Epic Games (Pharr 2005). It uses C++, and also has a code blocks system named blueprints, that allows users to prototype ideas really fast and without programming knowledge. It has the capabilities of producing photo realist 3D visuals, which is perfect for immersive experiences. The downside, not every workstation has the hardware to run the engine smoothly.

Unity is another example of a game engine accessible to the public and was introduced in 2005. Just one year after its release in 2006, it won an award at Apple's 2006 Worldwide Developer's Conference (AXON 2016). It uses C Sharp and has many plugins that allow users increase the engine features. Among those plugins we can find one that was created especially for Artificial Intelligence (AI), the Unity ML-Agents Toolkit, which is a great tool for creating realistic and complex AI environments to train models (Arthur Juliani 2018). Unity also has the advantage of being less demanding on the hardware, while still maintaining the capabilities of achieving photo realistic results.

As it was highlighted previously, game engines are perfect for developers who want to prototype game ideas fast, but they can also be used to create other kind of projects that nothing have to do with entertainment software. They can be used in the movie industry, projects like the movie Demonic by Neill Blomkamp are a perfect example of how such technology's can be used to expand the boundaries of what's been done before and what there is to explore (Failes 2021). Real world simulations can also be done, in the paper "Modeling Realistic 3D Trees Using Materials From Field Survey for Terrain Analysis of Tactical Training Center", the researchers developed a system that simulated a terrain with realistic trees, military units could then used the simulation to help them define a strategy to better prepare themselves (Robert Ornprapa P. 2019). Examples like this show us that game engines have transcend their main purpose, and are now a tool that is only limited by the users imagination and ingenuity.

2.6 Unity ML-Agents

ML-Agents is a unity open source toolkit library under the Apache 2.0 license. It gives the possibility of training intelligent agents in both games and simulated environments. Some of the machine learning methods it can use are RL, IL and neuro-evolution. They can be used to train the intelligent agents through simple Python API. It also provides the implementation of the most advanced algorithms so that game developers and hobbyist can train intelligent agents to use in their games. The agents can then be used for controlling non playable characters, automatically test game builds, and evaluating pre-released versions of different game design decisions (Jun LAI 2019).

ML-Agents contains five high-level components that can be seen in figure 2.5 and are as following:

The first is the learning environment which contains the unity scene and game characters. Unity provides the environment from which the agents will collect the observations, act, and learn. The unity scene should be built having in mind the goal the agent has to achieve. If the agent is being trained to operate in a complex game or simulation, its more efficient and practical to create different environments from which the agent will learn specific actions, before training in the final game or simulation complex environment. ML-Agents includes a Unity SDK that allows the transformation of any Unity scene into a learning environment by defining the agents and their behaviors (ML-Agents 2021a).

The second component is the Python low-level API which contains the low-level Python interface for the interacting and manipulating a learning environment. Unlike the learning environment the Python API is not part of Unity, it lives outside and communicates with Unity through the communicator. The API is in a dedicated *mlagents_env* Python package and is used by the Python training process to communicate with and control the training. However the API can also use Unity as the simulation engine for machine learning algorithms (ML-Agents 2021a).

The third is the external communicator, briefly referenced in the second component, it is responsible for connecting the learning environment with the Python Low-Level API, and it lives within the learning environment.

The forth is the Python trainer, which contains all the machine learning algorithms that enable the training agents. Those algorithms are implemented in Python and are part of their own *mlagents* Python package. The packages exposes a single command-line utility *mlagents-learn* that supports a couple of training methods and options. Python Trainers interface only with the Python Low-Level API (ML-Agents 2021a).

The fifth is the Gym Wrapper, it's not present in the figure 2.5. Usually machine learning researchers interact with simulation environments via a wrapper provided by OpenAI called gym. ML-Agents provides a gym wrapper in a dedicated *gym-unity* Python package that allows using it with existing machine learning algorithms that use gym (ML-Agents 2021a). Gym is an open source Python library used to develop and compare reinforcement learning algorithms and environments, as well as a standard set of environments compliant with that API. Gym's API has been the field standard since it has been released (OpenAI 2022).

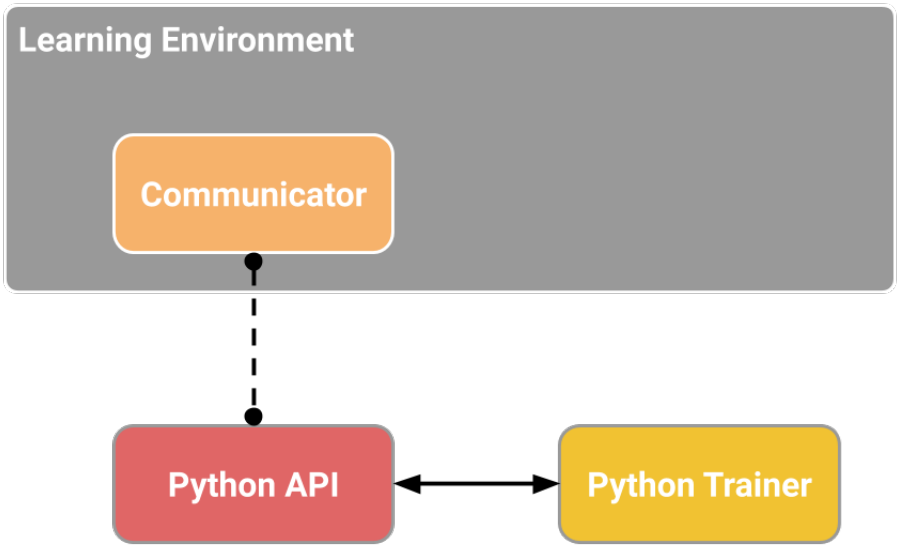

Figure 2.5: Simplified block diagram of ML-Agents (ML-Agents 2021a).

Exploring further the learning environment, it contains two Unity components to help organize the Unity scene. The first one are the agents, who are attached to a Unity GameObject (GameObjects are any kind of object present in the Unity scene), and handles generating its observations, performing actions received be it by inference or heuristic, and assigning positive and negative rewards. Agents are linked to the second Unity component, the behaviors, which define specific attributes of the agent such as the number of actions that the agent can take. Each behavior is identified by a *Behavior Name* field that needs to be unique. Behaviors can be seen as functions that receive observations and rewards from the agent and return actions. Behaviors can be one of three types: learning, heuristic or inference. A Learning Behavior is one that is not defined but about to be trained. A Heuristic Behavior is one that is defined by a hard-coded set of rules implemented in code. An Inference Behavior is one that includes a trained Neural Network file. In essence, after a Learning Behavior is trained, it becomes an Inference Behavior (ML-Agents 2021a).

Learning environments will always have one agent for every character in the scene. Each agent must be linked to a behavior, agents with similar observations and actions can have the same behavior, but different instances, this ca be seen in figure 2.6.

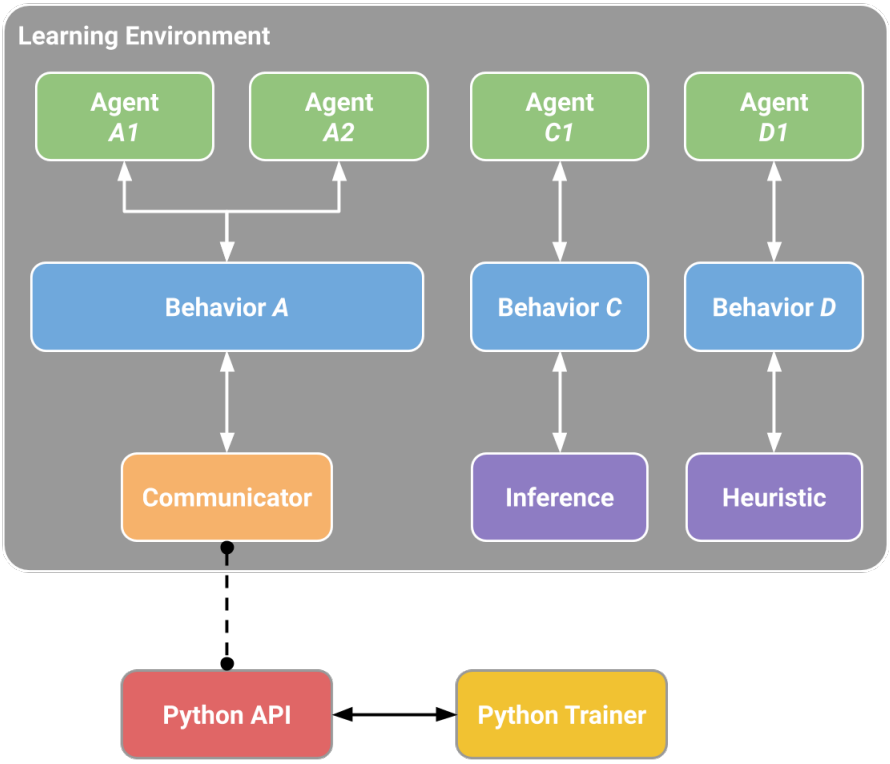

Figure 2.6: Example block diagram of ML-Agents Toolkit (ML-Agents 2021a).

2.7 Related Work

This section contains works in the field of RL, IL and habitat simulation, that helped define the course of action to develop this study.

2.7.1 Realistic animal behaviors in a virtual island

In this paper the authors created a virtual representation of the island of Chios located on the East Aegean Sea. The simulation was created with Unity 3D, which is a game engine that's been growing in popularity and features.

The authors chose to use FL to simulate the animals behavior. In fuzzy systems the uncertainty of the real world are met more appropriately due to the formation of soft or vague boundaries instead of crisp boundaries (Ertan Turan 2019).

In this simulation the animals have three possible behaviors, they can be passive, cautious or aggressive. Besides their behavior, they have a health system and a confidence level. By combining this three variables, when another animal enters their field of vision, they chose what seems to be the best option, fight or flight, or simply ignore the new presence.

2.7.2 Physical Habitat Simulation

In this study the authors created a physical habitat simulation, to predict the impact of weir removal on the composition of fish community in the river. The simulation site was a 900

meters long reach in the Gongneung-cheon Stream in Korea, at the middle of which the weir was located (Sung-Uk Choi 2018).

The simulation used ANFIS method, and the results were compared with those from a knowledge-based model, achieving better results than the latter. The results predicted correctly the change in the composition of fish community after the weir removal.

2.7.3 Machine Learning Algorithms in the Unity Environment

"Analysis of the possibilities for using machine learning algorithms in the Unity environment" is a paper that studied the applicability of machine learning on the Unity platform while using ML-Agents. Two algorithms were compared, PPO and Soft Actor-Critic and both algorithms results were improved with GAIL. The final results showed that PPO can perform better in uncomplicated environments with non-immediate rewards, and using GAIL improved the learning performance (Karina Litwynenko 2021).

2.7.4 Imitation Learning for a Custom Gameplay Using ML-agents

This paper begins by highlighting how hard it is to obtain a realistic behavior for non playable characters in a video game, and proposes the hypothesis that Unity with ML-Agents could be used to create the desired behavior, without it being explicitly written via code.

The method that was chosen was RL with IL, the work begins by recording a real person playing the game, collecting all the observations and actions. The results reported where accurate, stating that the agent playing style was the same as the player that recorded the training sessions, but when the agent was added to the environment it started moving slowly due to the environment graphical complexity (Amira E.Youssef 2019).

Chapter 3

Methods

This topic will discusses the methods and tools that were used to create the solution developed in this study.

3.1 Technology & Tools

3.1.1 Hardware & Software

The development of the present study was made in a desktop computer with the following specifications, a central processing unit (CPU) AMD Ryzen 7 2700 Eight-Core Processor 3.20 gigahertz (GHz), 16 gigabytes (GB) of random access memory (RAM) running at a frequency of 3200 megahertz (MHz), an M.2 solid-state driver (SSD) with a storage capacity of 500 gigabytes (GB) and a Nvidia RTX 2070 graphics processing unit (GPU) with 8 gigabytes (GB) of memory. This hardware might be slightly outdated, but it still performed decently when training neural networks and developing the training environments.

The used operating system (OS) was Windows 10 64-bit. This choice was made based on the processor that has a 64 bit architecture, and the chosen game engine runs smoothly and is well supported in Windows platforms.

3.2 Model Developed

The model was developed in Unity for a couple of reasons highlighted in the game engines section, it's light on the hardware, allowing developers to work remotely on low end machines. Prototyping ideas is easy and fast, while still giving room to improve the final result and polish it to high end quality. Has been used widely by professionals and academics alike which proves how versatile it can be. Most importantly it has a dedicated AI toolkit library called ML-Agents that allows the creation and usage of environments to train intelligent agents.

ML-Agents toolkit documentation recommends PPO algorithm stating that the method seems to work for most cases, and refers its stability (ML-Agents 2021a), also the paper in the subtopic 2.7.3 highlights how PPO performs better in uncomplicated environments, and when training agents to perform in complex scenarios, the training environments should be task oriented and uncomplicated, before increasing the difficulty of the task.

3.2.1 Environment

The agents are all trained in the same environment, a 3D aquarium that can be seen in the figure 3.1. The green lined boxes that are visible, exist all around the aquarium, and are there to prevent the intelligent agents from leaving the training area. If they try to leave the training area, a method is triggered when colliding with the green boxes, that method will destroy the agents and finish the training session, and spawn them inside starting a new training session. Ending the session this way, awards the intelligent agents a negative reward, as this is not the desired outcome from the training.

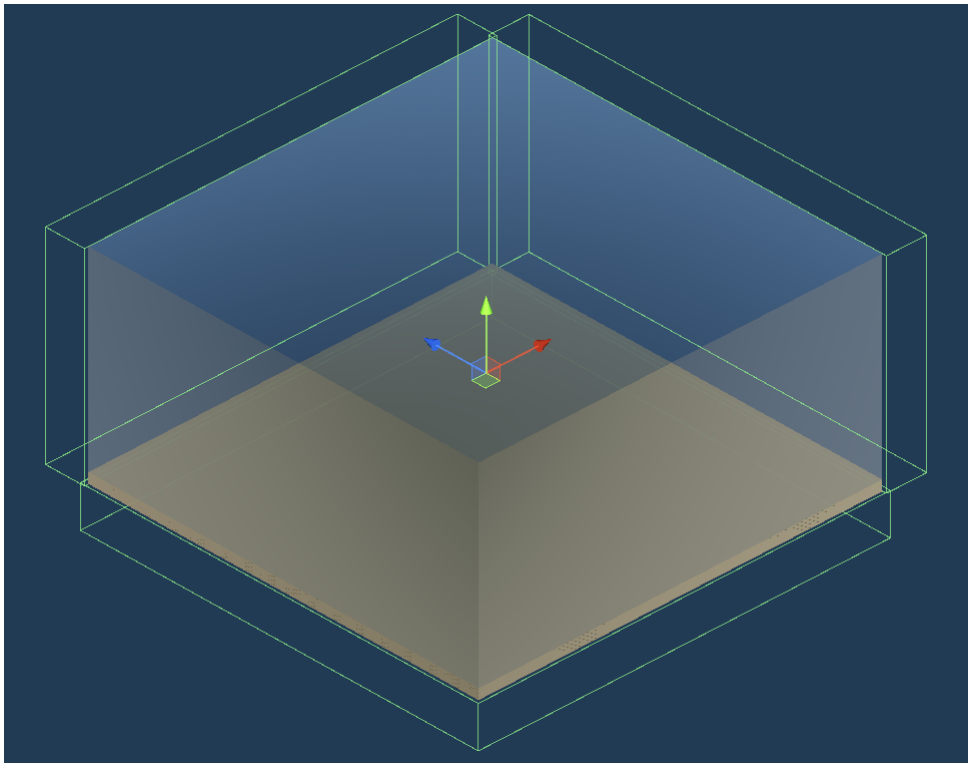

Figure 3.1: 3D aquarium environment used for the training.

The aquarium ground is responsible for spawning the seaweeds that the intelligent agents will be eating to survive. In order to spawn those seaweeds, it uses a object pooling script, its parameters can be seen in the figure 3.2. Using this object pooling script is very simple and user friendly. For the seaweed only matters the "Object To Pool" that is the object that will be spawned, the "Amount To Pool" represents how many objects should be spawned at a maximum, the "Spawn Area" defines the area where they can be spawn, to prevent objects being spawned outside the training area and the "Spawn Interval" sets the interval for the objects to spawn.

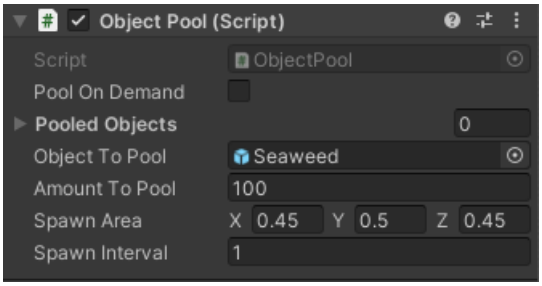

Figure 3.2: Object pooling script parameters.

Spawning the seaweeds randomly plays an important role in the intelligent agents training, if the training environment had a static object spawned in the same position training session after training session, the agent could be learning to move to that position instead of learning the goal is to collect the food. Figure 3.3 shows the desired result.

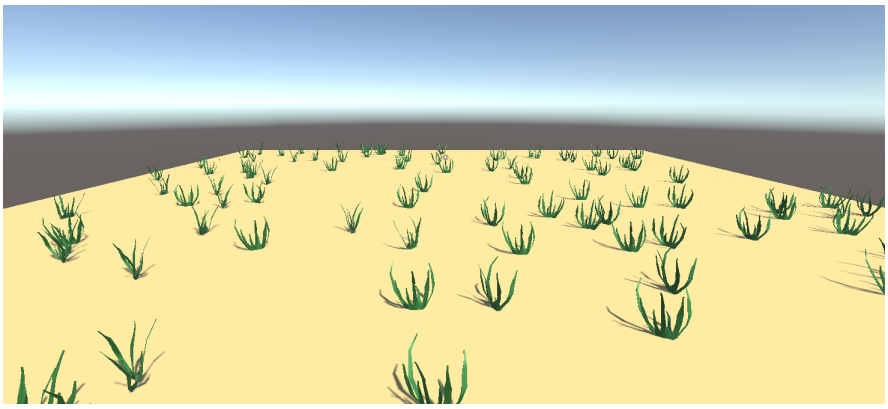

Figure 3.3: Aquarium with seaweeds randomly spawned.

Figure 3.4 has a close up of the seaweed 3D model, and the script attached to it. Since the seaweed is just an object used to help train the intelligent agents, it only needs the parameter "Caloric Value", that defines how many calories the agents get when eating the seaweed.

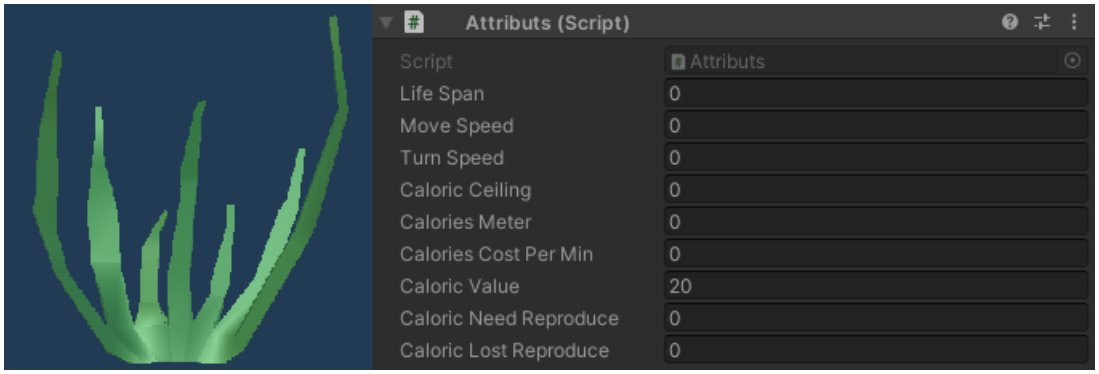

Figure 3.4: Seaweed 3D model and attributes.

3.2.2 Intelligent Agents

In this topic it will be discussed the intelligent agents that were developed in this study, the components they have, they're behaviors, the neural network used, reward system and training environment.

Components

The intelligent agents contain two important basic components, the box collider and the rigidbody, those are Unity components, that allow the intelligent agents to interact with the environment and detect collisions. Figure 3.5 shows an example of a intelligent agent inside its box collider on the left, and on the right it has the box collider and rigidbody components.

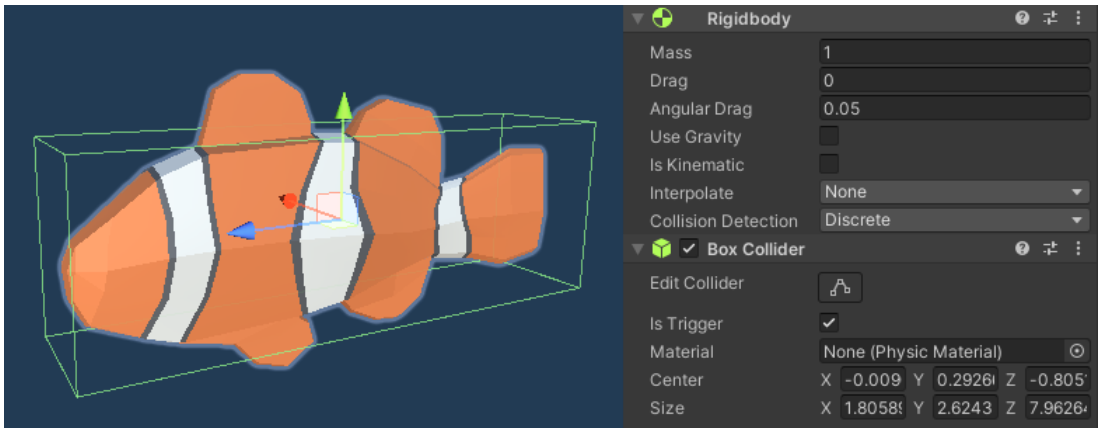

Figure 3.5: Intelligent agent box collider and rigidbody.

The intelligent agent attributes are set in the same script that can be seen in figure 3.3, those attributes play a role in the training sessions, all attributes will be explained bellow.

"Life Span" is how long the agent is active, after that time the agent will be destroyed, it serves two purposes, for the simulation it represents how long an animal can live, for the training sessions can be used to prevent the agent from procrastinating, because when the timer ends the agent will be destroyed and a negative reward will be given.

"Move Speed" defines how fast the agent can move itself and "Turn Speed" how fast he can change direction.

"Caloric Ceiling" is how many calories the agent can store. "Calories Meter" is how many calories the agent has to burn currently. "Caloric Value" is how many calories the agent gives if other agents eats it. "Caloric Need Reproduce" is how many calories the agent needs to have offspring's. "Caloric Lost Reproduce" is how many calories the agent loses when having offspring's.

"Calories Cost Per Min" is how many calories the agent loses per minute or second, depending on the configuration. It can be used to give a negative reward in the same amount of calories lost, preventing the agent from not taking action and staying still, and also simulate death by starvation.

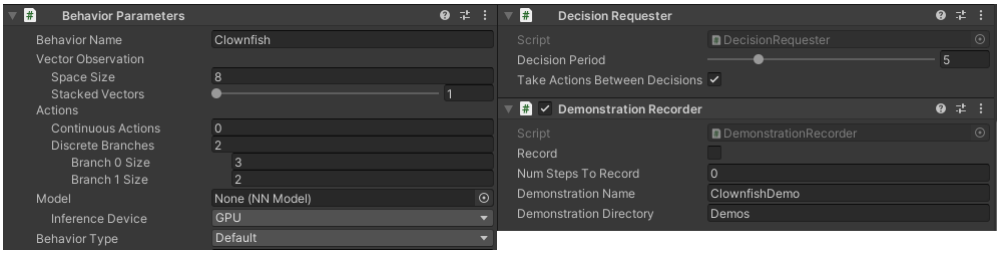

Figure 3.6: ML-Agents components.

The figure 3.6 shows the ML-Agents components used in the intelligent agents. In the "Behavior Parameters" we have the "Behavior Name" which needs to be the same as the neural network ID parameter.

"Vector Observations" defines how many observations we want to send to the neural network, for example an Boolean variable uses 1 position, a Vector3 since it is 3 coordinates (x, y, z) uses 3 positions. Observations should contain all the variables needed for the intelligent agent to solve the given problem.

"Actions" defines how many branch's of inputs the neural network can send to the intelligent agent. Continuous means that the values can be floats, while in discrete the values are integers. Each branch supports multiple inputs values, for example, in the figure 3.6 we can see the branch 0 with 3 inputs, those 3 inputs represent left right and no action, so the neural network can move the agent left right or not move it at all.

"Model" allows the agent to use a trained neural network, this is useful to use the intelligent agent in a simulation or game.

"Inference Device" picks the inference mode that will be used to train the model, it supports the following options. "Burst" inference, CPU using Burst, corresponds to CSharpBurst in Barracuda. "CPU" inference, corresponds to in CSharp Barracuda. "GPU" inference, corresponds to ComputePrecompiled in Barracuda.

"Behavior Type" defines from where the agent receives the inputs, if set as inference, it receives them from the neural network, its used when training the neural network. Heuristic option receives the inputs from the computer peripherals (keyboard, mouse, etc), its used to test the intelligent agent and environment while developing, and can be used in IL. Set as default it chooses between inference and heuristic depending on the first inputs it receives.

The "Decision Requester" component automatically request decisions for an intelligent agent instance at regular intervals. "Decision Period" is the number of steps an agent can take before requesting a decision. "Take Actions Between Decisions" indicates whether or not the agent will take an action before requesting a decision.

"Demonstration Recorder" component allows the developer to start recording training sessions, to be used in IL. "Record" whether or not to record the training session. "Num Steps To Record" sets the number of steps that the component will record, if the value is 0 it will record unlimited number of steps. "Demonstration Name" is the name the recording will have, and "Demonstration Directory" is the saving folder.

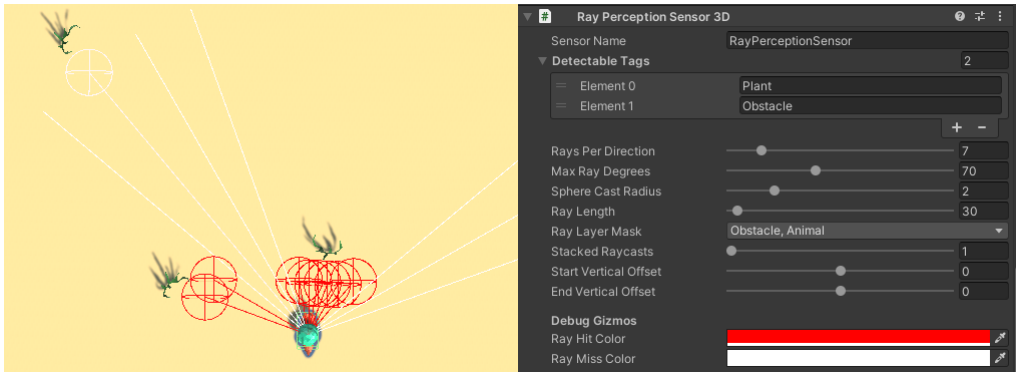

Figure 3.7: ML-Agents ray casting.

Figure 3.7 shows the "Ray Perception Sensor 3D" component. This component casts rays with spheres on the ending side. By using this rays the intelligent agent can observe the environment. "Detectable Tags" in Unity tags are a way to classify or categorize objects, this parameter defines which tags the rays will be able to detect, defining which objects the intelligent agent will be able to work with. "Ray Layer Mask" in unity the objects can be distributed across layers, this parameter defines which layers the rays will be able to detect objects, defining which layers the intelligent agent will be able to see. The rest of the parameters are essentially used to change the rays attributes, on the left side of the figure 3.7 we can see an intelligent agent casting rays and the rays hitting in the different objects.

Attributes

The intelligent agents attributes have been set for training purposes only, they do not represent any resemblance to the real world, and are not based in any real animal. Nonetheless all attributes can be changed by an expert if needed. In the figure 3.8 we can see the snail intelligent agent and its attributes. The figure 3.9 contains the clownfish intelligent agent and its attributes, it is inspired by the real world clownfish, but its not suppose to represent a realistic counterpart.

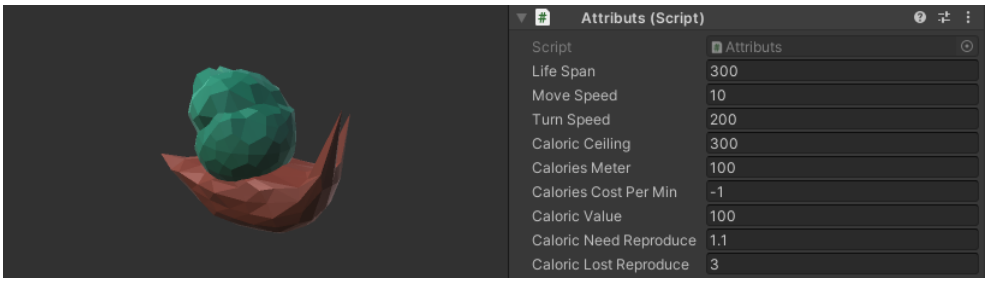

Figure 3.8: Intelligent agent snail 3D model and attributes.

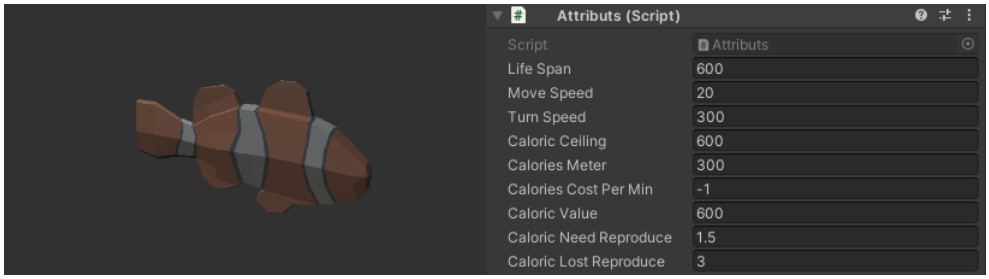

Figure 3.9: Intelligent agent clownfish 3D model and attributes.

Behavior

The snail behavior consists in eating the seaweed and increase its calories, when the snail has enough calories it doesn't need a mate to reproduce, it can use self fertilisation, this is based in something that snails can actually do in nature. The cycle will repeat until the snail life span ends.

In the clownfish behavior it also needs to ingest calories until it has the necessary amount to reproduce. What differs from the snail is that the clownfish can't use self fertilisation and has to find an anemone. First a female has to place the eggs in the anemone, and then the male can fertilize the eggs and the offspring's will spawn from the anemone. We can see an example of a clownfish next to an anemone in figure 3.10.

Figure 3.10: Clownfish agent swimming next to anemone.

Neural Network

```
behaviors:
  Clownfish:
    trainer_type: ppo
    hyperparameters:
      batch_size: 128
      buffer_size: 2048
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
      gail:
        strength: 0.1
        demo_path: Demos/ClownfishDemo.demo
    keep_checkpoints: 5
    max_steps: 500000
    time_horizon: 128
    summary_freq: 20000
    threaded: true
```

Listing 3.1: Yaml file used to configure the parameters of the neural network.

The YAML file 3.1 contains the configurations used to build the neural network. To understand the different parameters and what values would be more fitting for model developed, the ML-Agents documentation was used (ML-Agents 2021b). Right besides behavior the word "Clownfish" can be read, that means the neural network will be used for the clownfish behavior seen in the figure 3.6, if the name isn't the same, the model wont work properly. *trainer_type* was set to PPO, cause that's the RL algorithm that is being used.

keep_checkpoints are the maximum number of model checkpoints to keep and it was set as 5. *max_steps* are how many steps the agent must take in the environment before ending the training session, the value was left at 500000, past this value the training would return similar results and was just taking longer to end. *time_horizon* was set as 128, it controls how many steps of experience to collect per-agent before adding it to the experience buffer. *summary_freq* was set to 20000, it defines the number of experiences that needs to be collected before generating and displaying training statistics, the generate graphs can be seen in the Tensorboard. *threaded* was set as true, allow environments to step while updating the model, it speeds up the training.

The hyper parameters were set as follow:

batch_size is the number of experiences in each iteration of gradient descent, the documentation recommends PPO with discrete actions to be between 32 and 512, 128 was the value picked to begin with, and can be changed later on if the training results aren't satisfactory.

Next is the *buffer_size*, which is the number of experiences to collect before updating the policy model, for PPO the documentation recommends between 2048 and 409600. The buffer size should always be multiple times bigger than the *batch_size*, so the value as set to 2048 to begin with.

learning_rate corresponds to the strength of each gradient descent update step. It was left with the default value.

beta was also left with the default value, and is the strength of the entropy regularization, it makes the policy more random. Increasing this value increases the agent random actions.

epsilon was also left with the default value, it corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. Small values result in more stable updates and slow training process.

lambd was also left as default, is the regularization parameter (lambda) used when calculating the Generalized Advantage Estimate. It is how much the agent relies on its current value estimate when calculating an updated value estimate.

emph_epoch was left as 3, number of passes to make through the experience buffer when performing gradient descent optimization. It should increase as the *batch_size* is increased. Reducing the value ensures more stable updates but slower learning.

learning_rate schedule determines how the learning rate changes over time. The documentation recommends linear for PPO, so the value was set to linear. Linear means it will decay the learning rate linearly, reaching 0 at max steps.

Changing to the network settings, are as follow:

normalize was set with default value false, it controls whether normalization is applied to the vector observation inputs. This normalization can be helpful in cases with continuous control problems, but can be harmful with discrete control problems, since the model uses discrete actions, it was left as false.

hidden_units are the number of units in the hidden layers of the neural network. The more complex the problem, the more hidden units should be needed, default value is 128, but in this instance the value was set to 256.

num_layers are the number of hidden layers in the neural network. The value was set to 2, since those layers are enough for the problem that's being solved, and increasing that number would increase the training time and reduce its efficiency.

vis_encode_type is configured as simple. This is used to define the encoder type for encoding visual observations, it uses a simple encoder which consists if two convolutional layers. Due to the size of the convolutional kernel, the observations size can't be lower than 20x20.

At last the reward signals are the following ones:

The extrinsic rewards *strength* was left as default 1.0, it's the factor by which to multiply the reward given by the environment. The extrinsic *gamma* was left as 0.99, it's the discount factor for future rewards coming from the environment. Since the agent has to take immediate actions, its value should be bellow 1.

The GAIL intrinsic reward enables the use of the recorded training sessions to be used in IL. The *demo_path* is the path of the *.demo* file with the recorded training sessions. The

strength was initially set to 0.9, and gradually reduced to 0.1, it defines how much the training will mimic the demos, in the begin the agent should mimic the demo until it is trained for the task, and after that the value can be reduced to allow the agent to surpass the human that recorded the training sessions.

Rewards

Achieving the correct value for the rewards, was a work of planning and trial by error. Positive rewards will be discussed first with code examples.

When the intelligent agent is spawned, the first steps should be towards getting food, so every time the intelligent agent gathers food, the ChangeCalories method that can be seen in the code snippet 3.2 is invoked. The amount of the reward is the amount of calories eaten divided by 10, doing this normalizes the value and keeps the reward value under control. This way the model is scalable, and if more sources of food were to be added, the agent could pick between them.

```
    /// <summary>
    /// Change the calories value according the given caloriesVal
    /// </summary>
    public float ChangeCalories(float caloriesVal)
    {
        AddReward(caloriesVal / 10);
        caloriesMeter = caloriesMeter + caloriesVal;
        caloriesMeter = caloriesMeter <= caloricCeiling ? caloriesMeter
    : caloricCeiling;
        return caloriesMeter;
    }
```

Listing 3.2: Change calories method.

When the snail intelligent agent reaches the necessary amount of calories he has to reproduce, when reproducing successfully it will be given a reward of 100. The clownfish intelligent agent on the other hand, gets a reward of 200, since he can process the double of the calories from the same food source.

All the positive rewards are covered. The negative rewards, the ones responsible from preventing the intelligent agents from taking the wrong actions, will be explained bellow.

To force the agent to take action from the beginning, negative rewards are given as time passes, the method responsible for that can be seen in 3.3. The amount of negative reward is equal to the attribute "Calories Cost Per Min" seen in figure 3.9.

```
/// <summary>
/// Burn the calories cost per minute
/// </summary>
private void BurnCalories()
{
    float calories = ChangeCalories(caloriesCostPerMin);

    if (calories <= 0)
    {
        Die();
    }
}
```

Listing 3.3: Burning calories method.

Any action that kills the intelligent agent will invoke the method 3.4, giving a negative reward of -50. Currently those actions include trying to leave the training environment and not consuming enough calories. More situations could easily be added, by simply detecting the situation and invoking the method.

```
/// <summary>
/// Function to run when the agent "dies"
/// </summary>
public void Die()
{
    EndTraining(failure, -50);
}
```

Listing 3.4: Die method.

When trying to reproduce without enough calories the snail intelligent agent receives a negative reward of -100. The clownfish agent on the other hand, receives a negative reward of -200 for the same reason explained before, receiving the double of the calories from the same food sources.

Observations

ML-agents supports observations vectors that were briefly discussed in topic 3.2.2 and can be seen in figure 3.6. The decisions the intelligent agent has to take, are based on the observations it takes from the environment, but some variables needed for those decisions, might not be in the environment, or might not be visible for the agent at a given moment. That's where observation vector can help, the developers can use them to send relevant information to the intelligent agent, that it wouldn't be able to gather any other way. The observations needed for the project will be explained bellow.

The intelligent snail agent observations method can be seen in the code snippet 3.5. The intelligent agent needs to know when its a good time to reproduce, and that information can't be gathered from observing the environment. The caloric value needs to be sent to the intelligent agent, but instead of sending the amount of calories, its better to send a Boolean, the model receives the value true or false and takes a decision, it won't even know what the true or false represent, but it will take actions depending on the value, and get the reward accordingly, and learn from it.

```
    public override void CollectObservations(VectorSensor sensor)
    {
        canReproduce = caloriesMeter >= caloricCeiling /
    caloricNeedReproduce;
        sensor.AddObservation(canReproduce ? 1 : 0);
    }
```

Listing 3.5: Snail observations method.

The clownfish intelligent agent, also needs the observation containing the Boolean signaling if it can reproduce or not. But besides that it needs a couple more information's. Since the it cant reproduce by itself and besides that needs to place/fecund its eggs in an anemone. That said the intelligent agent needs 3 more observations in order to remember where the anemone is if he strays too far. The distance to the anemone, the direction to the anemone and the direction it is facing, this can be seen in the method 3.6.

```
    public override void CollectObservations(VectorSensor sensor)
    {
        canReproduce = caloriesMeter >= caloricCeiling /
    caloricNeedReproduce;

        gameObject.tag = canReproduce ? "ClownfishFull" : "Clownfish";

        sensor.AddObservation(canReproduce ? 1 : 0);

        sensor.AddObservation(Vector3.Distance(hideOut.transform.
    localPosition, transform.localPosition));

        sensor.AddObservation((hideOut.transform.localPosition -
    transform.localPosition).normalized);

        sensor.AddObservation(transform.forward);
    }
```

Listing 3.6: Clownfish observations method.

Training PPO With Ray Casting

To train the agents ML-Agents provides two important methods, the OnEpisodeBegin and the EndEpisode, that can be seen in code snippet 3.7 and 3.8 respectively. OnEpisodeBegin method is invoke every time a new training session begins, and is useful to randomize the agent spawn location all its attributes and randomize the agent shelter location if it has one, this will prevent the agents from learning bad behaviors focused in going to a specific coordinate instead of completing a specific objective like eating or reproducing. EndTraining method was created to end the training session, give the reward, and give a visual feedback to the developers and experts about how the training session went. EndEpisode method belongs to the ML-Agents API and that's what actually ends the training session.

```
/// <summary>
/// ML-Agents overrides
/// </summary>
public override void OnEpisodeBegin()
{
    CancelInvoke();

    caloriesMeter = defaultCaloricMeter;
    transform.rotation = Quaternion.identity;

    if (training)
    {
        float x = Random.Range(-spawnArea.x, spawnArea.x);
        float y = spawnArea.y;
        float z = Random.Range(-spawnArea.z, spawnArea.z);
        transform.localPosition = new Vector3(x, y, z);

        if (hideOut != null)
        {
            x = x > 0 ? x + 15 : x - 15;
            z = z > 0 ? z + 15 : z - 15;
            shelterSpawnArea = new Vector3(x, y, z);
            hideOut.transform.localPosition = new Vector3(x, y, z);
        }
    }

    Invoke("Die", lifeSpan);
    InvokeRepeating("BurnCalories", 1.0f, 1.0f);
}
```

Listing 3.7: ML-Agents method OnEpisodeBegin overwritten.

```
/// <summary>
/// End episode, change environment material and add reward
/// </summary>
public void EndTraining(Material result, float reward)
{
    AddReward(reward);

    if (training)
    {
        environmentBase.GetComponent<MeshRenderer>().material = result;
        EndEpisode();
    }
    else
    {
        gameObject.SetActive(false);
    }
}
```

Listing 3.8: EndTraining method invokes EndEpisode ML-Agents method.

The training environment for the snails contains the aquarium that was previously developed, and one snail in it. The snail has to adopt the pretended behavior explained in the previous topic. When the training session is a success the ground remains yellow, if the intelligent agent fails, the ground turns red. This visual effect allows the developers to detect problems, for example if every single intelligent agent is succeeding and only one is failing, it could be due to a specific reason in the environment that can be explored to fix the problem.

With the training environment ready, the training sessions were recorded to be used later on with GAIL in IL. The result of those demonstrations can be seen in figure 3.11. It has the meta data, it contains the number of steps taken, the number of training sessions recorded and the mean reward value, a good demo will have only the necessary amount of reward for the desired behavior to be a success. It also contains the observations and the discrete actions it can receive. More and better information in the demos will improve the training success greatly.

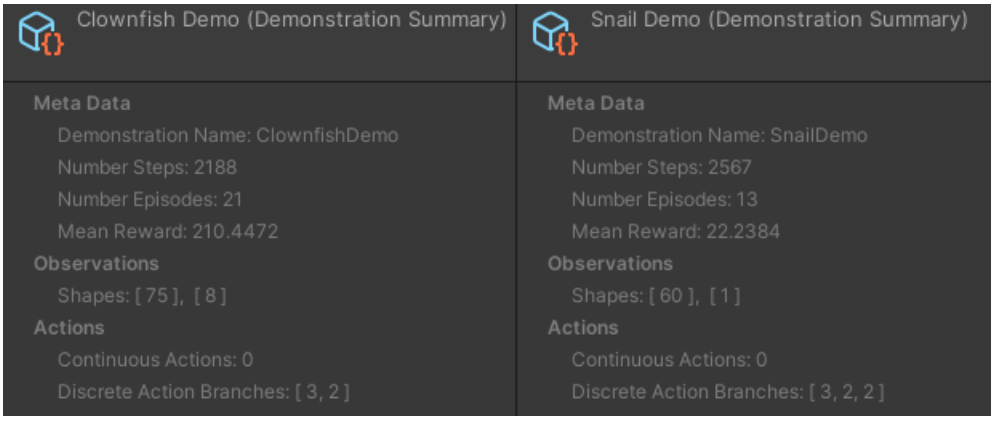

Figure 3.11: Snail and clownfish recorded demos.

With the environment and the demos ready to be used, the intelligent agents could start training. In the figure 3.12 we can see multiple intelligent agents training. The chosen inference device was GPU, and 9 environments at once was the maximum the desktop could handle without heavy stuttering.

The clownfish has all the same training characteristics discuss in the last paragraph, in addition to the anemone seen in the figure 3.10, that spawns in a different place each training season, otherwise the intelligent agent would be learning to move to a specific location instead moving towards the anemone.

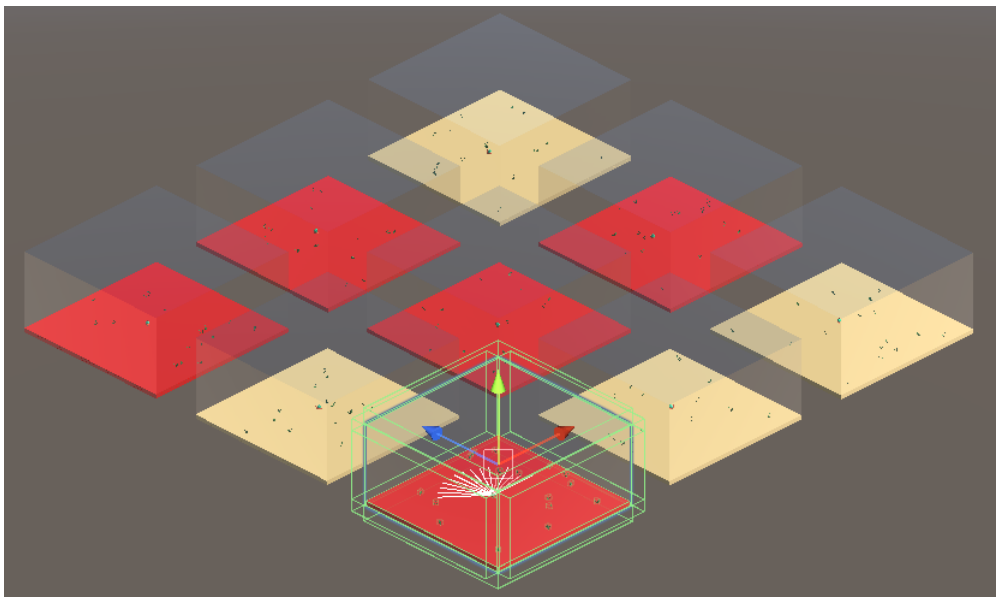

Figure 3.12: Multiple snails training, each with its own environment.

The snail was the first intelligent agent to be trained, since its behavior was more simple when compared to the clownfish. Two training sessions were made, one with RL and one with IL, the results can be seen in figure 3.13. This two training sessions had several problems that will be explained later on, but to notice how the IL demo achieved higher rewards in less time, it shows the potential in the technology.

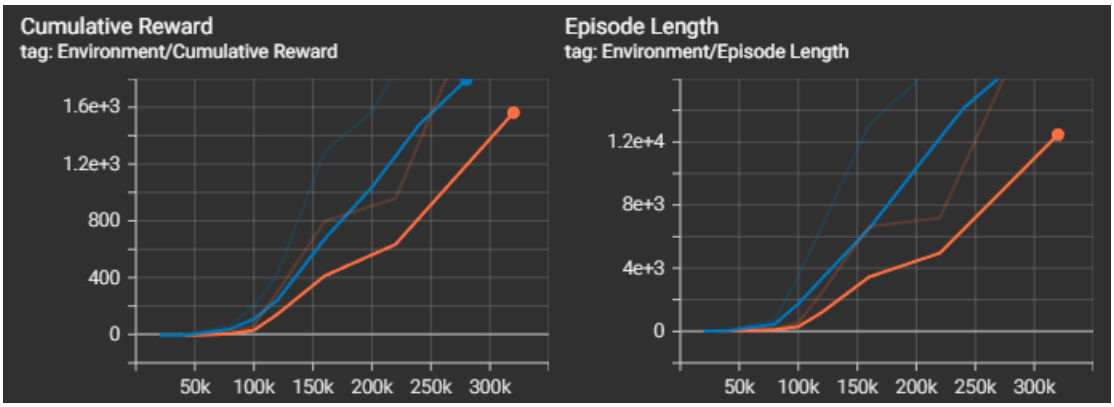

Figure 3.13: Snail training IL (blue) vs RL (orange).

Figure 3.13 training sessions had two major problems in the intelligent agent, the intelligent agent had no observation to notify it when he could reproduce or not, which was the main objective of the training session, and the reward for trying to reproduce at the time was too low, making the intelligent agent avoiding the action in search of bigger rewards.

After the observations were added and the rewards for successfully reproducing updated, a new demo with training sessions had to be recorded, so the agent could have that additional information when training. Figure 3.14 contains three training sessions. The red line one was the first, with GAIL extrinsic strength of 0.9. In the blue line the GAIL was reduced to 0.5. In the pink line the GAIL was reduced to 0.1. Its possible to see the reward value got more stable, and the episode length reduced greatly, meaning the intelligent agent was achievement the objective faster and efficiently.

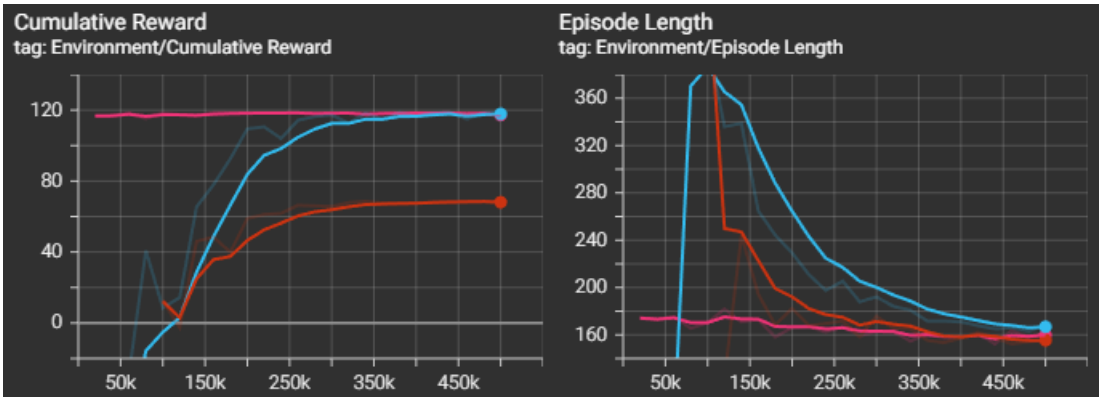

Figure 3.14: Snail IL improved.

When it came to the clownfish many more training sessions were required, as can be seen in figure 3.15, most of the training sessions have a reward value bellow 0, the intelligent agent

was managing to collect food, but not to reproduce, cause it wasn't finding the anemone most of the times.

After some research it was clear the agent needed some way to remember where the anemone was if he strayed too far away. So new observations were added, the direction the anemone was, the distance to it, and the direction the agent was facing. A new demo was recorded with this new observations, the GAIL extrinsic strength was set to 0.5 in the first new training session, and the intelligent agent start achieving more satisfactory results. Those results can be seen in 3.15, the two lines above 0. The second training session with the new observations was done with GAIL extrinsic strength of 0.1, showing the intelligent agent achieving a more stable reward value, and finishing the tasks way faster.

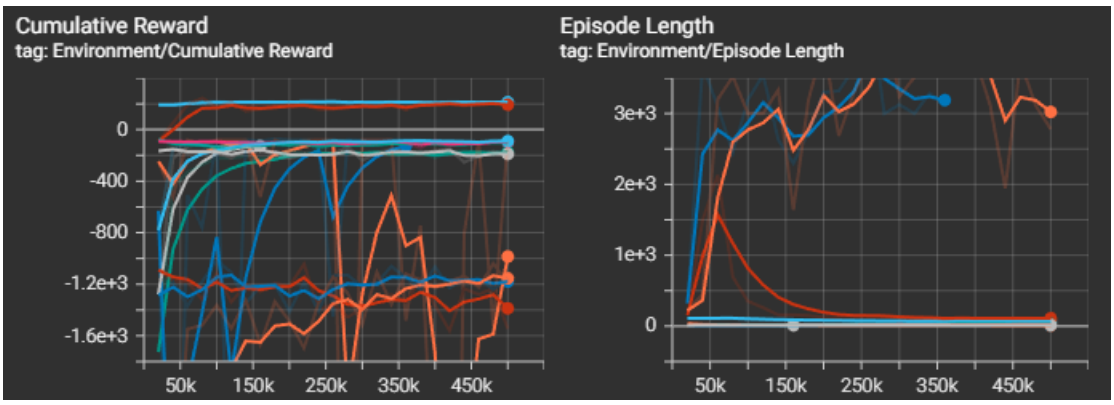

Figure 3.15: Clownfish IL training sessions.

Training PPO With Visual Sensor

As it could be seen in the previous topic, PPO with ray casting sensors works great for training intelligent agents to simulate the behavior of animals, but there is one weakness from the point of view of fidelity. Using ray casting sensor seen in figure 3.7 to detect the other objects, is a great advantage since there's no deceiving, the agent can clearly identify everything and know exactly what they are. In nature some animals use vision, there's different type of vision and different type of sensors for those who don't have sight, but some sort of vision is a pretty common sense in nature.

In order to explore a solution based in visual observations, the snail intelligent agent will be configured to do so. ML-Agents parameter *vis_encode_type*, explained in 3.2.2, allows the use of two CNN to encode visual observations.

Besides the parameter set to the value simple, the agent needs a couple of changes as well, to begin with the agent needs a camera sensor to capture the images that will be encoded by the CNN, figure 3.16 shows the camera sensor component. The width and height parameter are for the number of pixels in each direction. Gray scale is left unchecked since the images need to have color. Observation stack is set to 1, if the agent had to track moving objects that number had to be much higher. Observation type is left as default, the other option would be to set it as goal signal, that would collect observations containing goal information. Compression is set to PNG, and its the compression that will be applied to the generated images.

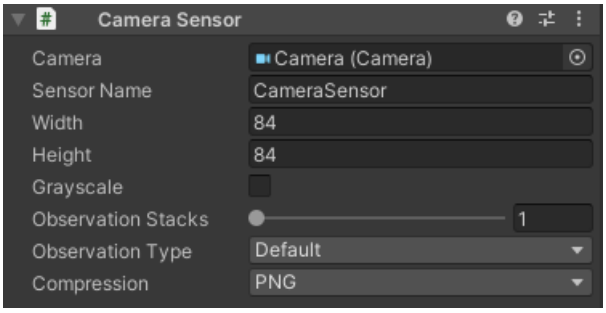

Figure 3.16: Snail camera component.

The environment had to change as well, since the agent uses visual observations and the environment boundaries are invisible, the agent wont be able to detect them and understand how far it can go. Figure 3.17 shows the same scene rendered from a normal camera on the left, and on the middle its possible to see the same scene, but from the intelligent agent perspective, the differences are on the walls that are now blue instead invisible, and the seaweed is a big green cube. Both changes are important to help the agent and reduce the training time. On the right we have a third example of the same scene, and its an example of image that's sent to the CNN for encoding, sending low resolution images speeds the process greatly, the objective is to reduce the resolution the maximum possible, while keeping it with enough information for the intelligent agent to use.

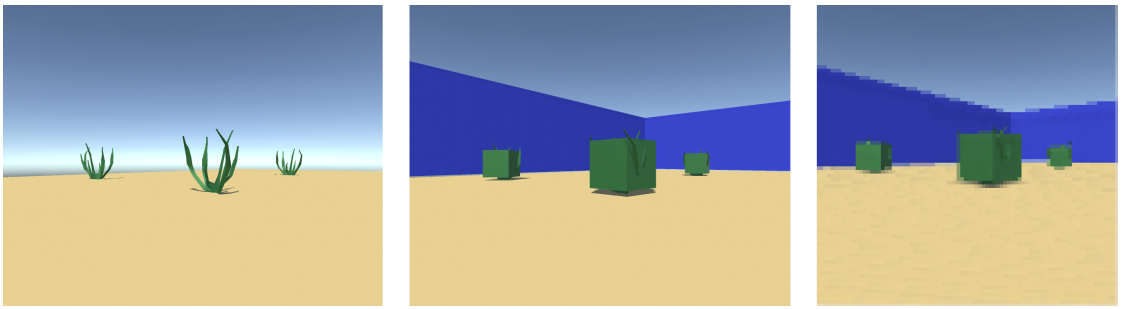

Figure 3.17: Normal camera left. Agent camera center. PNG sent to the CNN right. All representing the same scene.

To use IL with this new intelligent agent, a new demo had to be recorded, since the previous demo used different sensors the information in it isn't exactly the same. Figure 3.18 has a demo recorded with camera sensor.

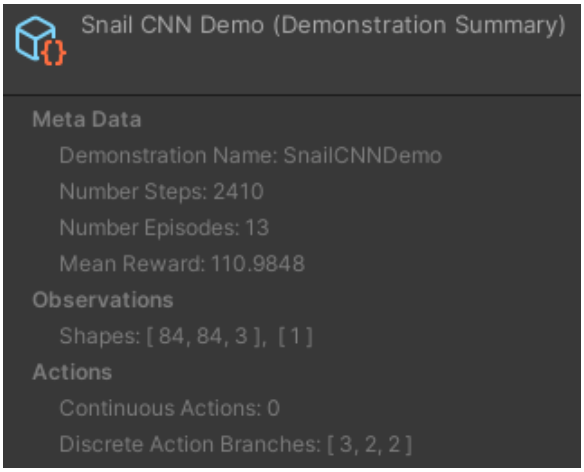

Figure 3.18: Demo with the intelligent agent using camera sensor.

Figure 3.19 show's the first two training's with the new recorded demo. In the first training the GAIL strength was set to 0.5, and in the second training the GAIL strength was reduced to 0.1. In the first episode, the length was increasing with time, due to the quality of the demo. Reducing the GAIL strength in the second demo, made the intelligent agent look less to the demo, and improved over the mistakes that were done while recording the demo, which decrease greatly the time the agent took to achieve the goal. The mistakes that happened when recording the demo, where based on the difficulty in controlling the agent with a first person camera. This highlights two important topics to have in mind when developing such solution. The first one is that when using IL, is really important to record sessions with quality. Second its about the camera and controls that are used to interact with the agent, if the controls are not user friendly, the expert may not be able to recreate the behavior he desires.

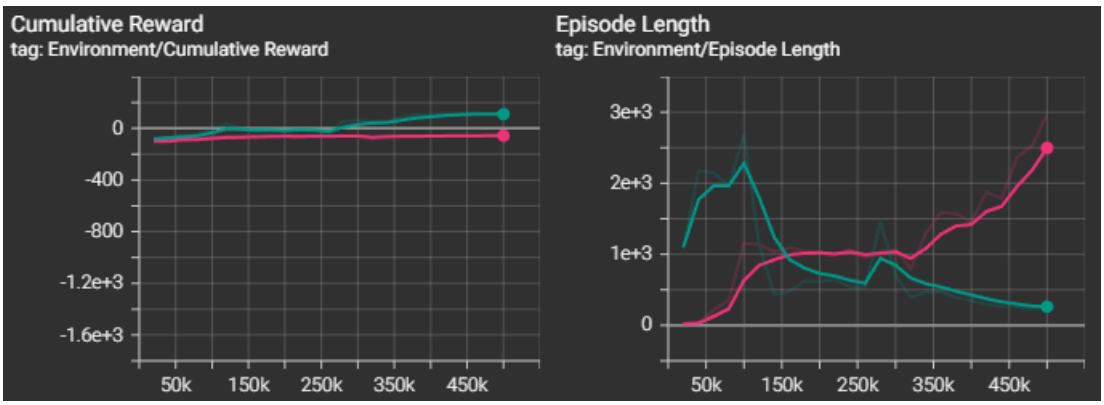

Figure 3.19: First and second training with camera sensor and CNN.

Figure 3.20 contains the second and third training session. The second session by the end had already almost reach the desired behavior, but it could be improved, so the third training session was configured to improve over the second, but without GAIL. Which give the intelligent agent the ability to go straight to the goal without external noise from the demo and reduce the episode length greatly.

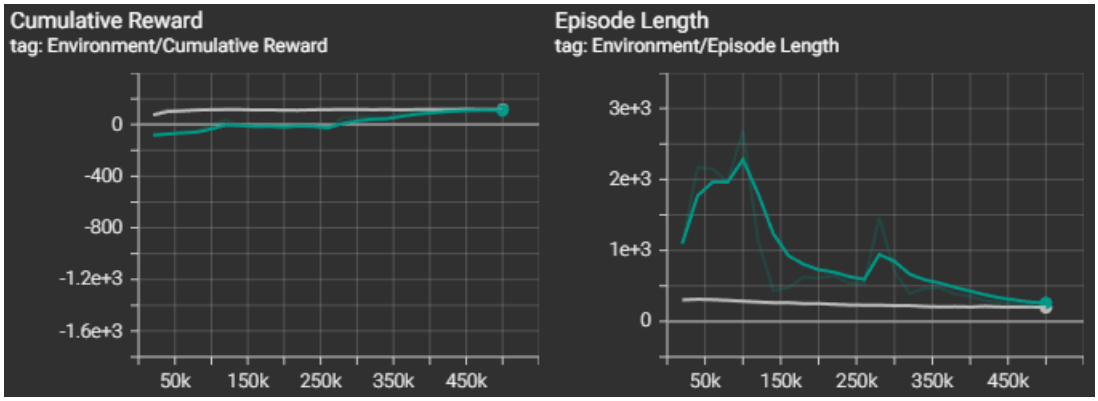

Figure 3.20: Second and third training with camera sensor and CNN.

3.2.3 Results

To test the intelligent agents that have been trained, a new environment was created, containing the seaweeds and anemones. The objective is to simulate what would happen if the clownfish would invade the snail environment. Figure 3.21 is a screenshot taken during the experiment.

Figure 3.21: Experimenting with clownfish and snail intelligent agents competing for food.

Before spawning the agents, the seaweeds were left alone to populate to a maximum of 800 individuals, looking at graphic 3.22 its possible to see the seaweeds took around 110

seconds to reach the 800 individuals milestone, with no predators. By the time 4 snails were spawned in the environment, the seaweed population start dropping. When 4 clownfish were introduced, it dropped drastically. Looking only to the seaweeds for now its clear that since the predators were introduced it dropped to less then 2.5 percent of its peak, but then it remained with a stable population.

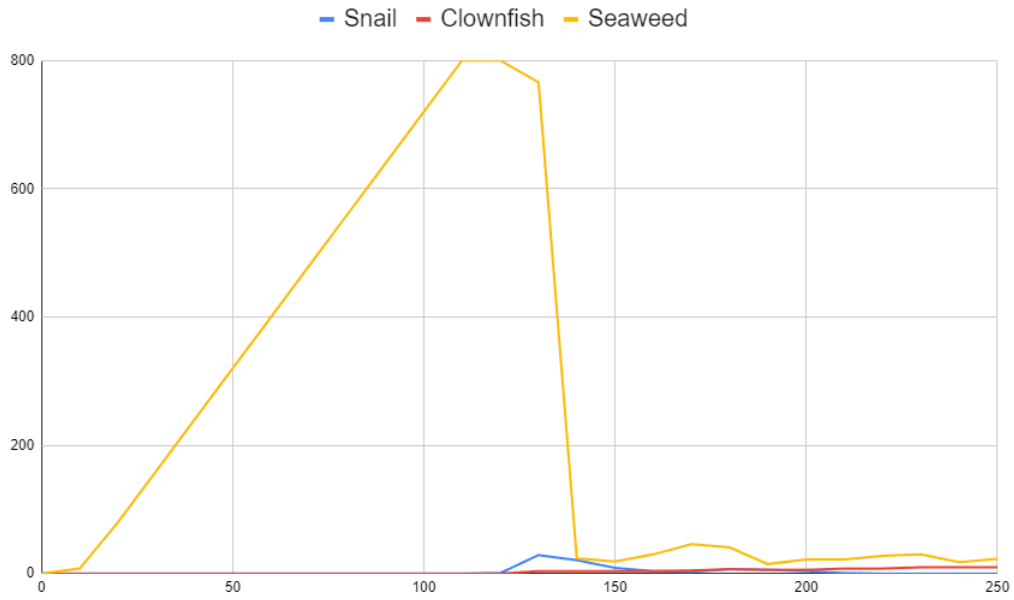

Figure 3.22: Number of snail, clownfish and seaweeds during the experiment, Y axis is for total number of agents, X axis is for time passed in seconds.

Looking at graphic 3.23, its possible to see a zoomed version of the graphic 3.22, and its more clear what happened between the snail and the clownfish. The snails jumped to almost 30 individuals, but by the time the clownfish were added, its population start dropping drastically. It then raises again and eventually it dropped to 0 individuals while the clownfish eventually seemed to hit a stable population for the available food sources.

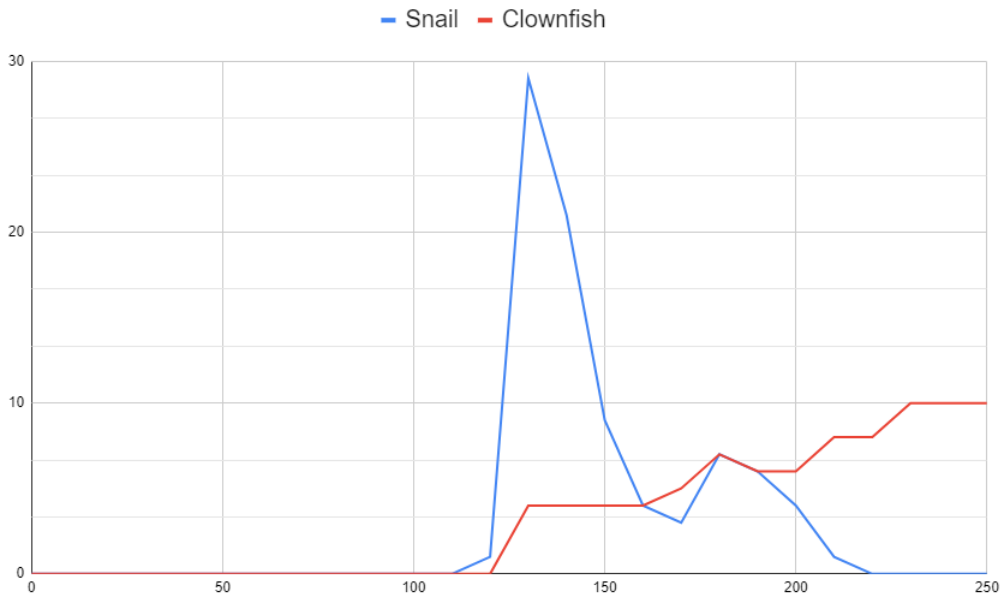

Figure 3.23: Zoomed version of the graphic 3.22 containing only the snail and clownfish.

This experiment was not planned and the agents attributes were given randomly. But the results is what matters, cause it shows the potential of this study. By looking at the graphics and the attributes its easy to conclude that the snails population dropped due to the food sources dwelling when they start eating it too much and reproducing too fast. Also the clownfish were added, and they move faster than the snails, and started competing with them for food. The snails had the upper hand when reproducing since they didn't need a mate, but that wasn't enough cause they were failing to get food, and as the clownfish population rose, the snails had more and more competition, until they went instinct in the environment.

Changing some variables, or changing the environment, would change the result, and that's the purpose of this study, while not a final solution, show whats possible, and is flexible enough to try different scenarios.

Chapter 4

Conclusion And Future Work

This study had as main goal to see if it was possible to use intelligent agents based on animals, to simulate the outcome of different species colliding with each others in a given habitat. The results were positive, and show what could be possible to achieve in a more structured and well financed project. It also helped to highlight some of the problems that appear while developing such intelligent agents. The next topics will discuss all this in more detail.

4.1 Contributions

The work developed in this study can help other students or professionals to understand the different tools ML-Agents has, and how to use them in order to teach a desired behavior into an intelligent agent. It also explains how to set up and configure a YAML file, containing all the necessary parameters, what each one is and does. Different neural networks were used, the process for using both of them was explained, the advantages and short comings are well documented, making it simple to understand the strengths and weakness and making it easy to choose between them. But mainly it shows that it is possible to use the virtual world, to simulate the natural world, in a more safe inexpensive and ethical way.

4.2 Validation of the Research Hypotheses

The first question was the following, can experts create a realistic behaviors by using RL with learning by imitation?

It was possible to show that IL is a very powerful tool, topics 3.2.2 and 3.2.2 show that's possible to create even a cooperation between two agents to reproduce, more complex behaviors can be achieved and experts are really free to explore as far as they want.

The second question was if Can intelligent agents interactions be used to predict changes in a given habitat population?

Topic 3.2.3 shows a experiment developed with the previously trained intelligent agents, and the result and information left after the experiment can be diagnosed and the reasons for the outcome can be study and addressed, the more complex and the more agents interacting with each other in the simulation, the harder is to predict what would happen, but this example with two agents, shows whats possible.

4.3 Final Remarks and Future Work Considerations

The study was satisfactory to research the hypotheses that were suggested, and it give a glimpse of what can be achieved in the future. In topic 3.2.2 its possible to see an example of the same snail intelligent agent created in the topic 3.2.2, but with visual sensors, this opens the possibility to developed solutions where the agents can camouflage themselves, and having agents camouflaging successfully and agents detecting those camouflages could be a whole new study. There are many animal attributes missing, adding those attributes would give more realistic simulation and increase the model complexity greatly. This are just some examples of what can be done to improve this study, there are many more, since the model is nature itself.

Besides predicting nature this study could be apply to video games. Some video games have heavy focus in simulation, its a genre in itself. When speaking about simulations, it could also be used to create realistic virtual reality experiences for users who may not be able to have access to them, for example, if a specific habitat is close to the public due to conservation issues, a virtual reality solution would give the second best option to experience it. The future should go through simulating everything that's possible and avoid unnecessary cost and loss in nature.

Bibliography

Abraham, A. (2005). "Adaptation of Fuzzy Inference System Using Neural Learning". In.

ML-Agents (2021a). *ML-Agents Toolkit Overview*. url: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md. (accessed: 10.10.2022).

– (2021b). *Training Configuration File*. url: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md. (accessed: 10.10.2022).

Ahire, Jayesh Bapu (2018). *The Artificial Neural Networks handbook: Part 1*. url: https://www.datasciencecentral.com/the-artificial-neural-networks-handbook-part-1/. (accessed: 10.10.2022).

Amira E.Youssef, Sohaila El Missiry (2019). "Building your kingdom Imitation Learning for a Custom Gameplay Using Unity ML-agents". In: doi: 10.1109/IEMCON.2019.8936134.

Arthur Juliani, Vincent-Pierre Berges (2018). *Unity: A General Platform for Intelligent Agents*. Ithaca. doi: 1809.02627. url: https://arxiv.org/abs/1809.02627.

AXON, SAMUEL (2016). *Unity at 10: For better—or worse—game development has never been easier*. url: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/. (accessed: 08.01.2022).

Babuška, Robert (1998). *Fuzzy Modeling For Control*. Springer. isbn: 978-9401060400.

Bhatt, Shweta (2018). *5 Things You Need to Know about Reinforcement Learning*. url: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html. (accessed: 10.10.2022).

Bojarski M. Testa D.D., Dworakowski D. (2016). *End to End Learning for Self-Driving Cars*. doi: 1604.07316. url: https://arxiv.org/abs/1604.07316.

D. H. Hubel, T. N. Wiesel (1959). "Receptive fields of single neurones in the cat's striate cortex". In.

Darwin, Charles (1859). *On the Origin of Species*. Empire Books. isbn: 978-1619491304.

Erik Brynjolfsson, Tom Mitchell (2017). "What can machine learning do? Workforce implications". In.

Ertan Turan, Gürcan Çetin (2019). "Using artificial intelligence for modeling of the realistic animal behaviors in a virtual island". In.

Failes, Ian (2021). *Neill Blomkamp on what Unity's volumetric capture brought to his new horror film, Demonic*. url: https://blog.unity.com/entertainment/neill-blomkamp-on-what-unitys-volumetric-capture-brought-to-his-new-horror-film. (accessed: 08.01.2022).

Federation, The National Wildlife (2020). *Invasive species—they may not sound very threatening, but these invaders, large and small, have devastating effects on wildlife*. url: https://www.nwf.org/Educational-Resources/Wildlife-Guide/Threats-to-Wildlife/Invasive-Species. (accessed: 08.01.2022).

Hajek, Petr (2010). *Fuzzy Logic*. url: https://plato.stanford.edu/archives/fall2016/entries/logic-fuzzy/. (accessed: 20.01.2022).

J. A. Bradshaw K. J. Carden, D. Riordan (1991). "Ecological applications using a novel expert system shell". In.

Jang, Jyh-Shing R. (1991). "Fuzzy Modeling Using Generalized Neural Networks and Kalman Filter Algorithm". In.

Jiang Hua, Liangcai Zeng (2021). "Learning for a Robot: Deep Reinforcement Learning, Imitation Learning, Transfer Learning". In: doi: 10.3390/s21041278.

John Schulman, Oleg Klimov (2017). *Proximal Policy Optimization*. url: https://openai.com/blog/openai-baselines-ppo/. (accessed: 29.01.2022).

Jun LAI Xi-liang CHEN, Xue-zhen ZHANG (2019). "Training an Agent for Third-person Shooter Game Using Unity ML-Agents". In.

Jyh-Shing Roger Jang Chuen-Tsai Sun, Eiji Mizutani (1997). *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Pearson College Div. isbn: 978-0132610667.

Karina Litwynenko, Małgorzata Plechawska-Wójcik (2021). "Analysis of the possibilities for using machine learning algorithms in the Unity environment". In: doi: https://doi.org/10.35784/jcsi.2680.

Kleene, S. C. (1956). "Representation of Events in Nerve Nets and Finite Automata". In.

Klimasauskas, CC (1989). "The 1989 Neuro Computing Bibliography". In.

Kuefler A., Morton J. (2017). *Imitating driver behavior with generative adversarial networks*. doi: 1701.06699. url: https://arxiv.org/abs/1701.06699.

Kumar V. Gupta A., Todorov E. (2016). *Learning Dexterous Manipulation Policies from Experience and Imitation*. doi: 1611.05095. url: https://arxiv.org/abs/1611.05095.

Laboratory, Pacific Northwest National (2021). *Neural Networks*. url: http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html. (accessed: 08.01.2019).

Lawrence, Jeannette (1994). *Introduction To Neural Networks: Design, Theory, and Applications, Sixth Edition 6th Edition*. California Scientific Software. isbn: 978-1883157005.

Mahajan, Chetanya (2020). "Reinforcement Learning Game Training : A Brief Intuitive". In.

Merel J., Tassa Y. (2017). *Learning human behaviors from motion capture by adversarial imitation*. doi: 1707.02201. url: https://arxiv.org/abs/1707.02201.

Miodrag Petkovic Ilija Basicevic, Dragan Kukolj (2018). "Evaluation of Takagi-Sugeno-Kang Fuzzy Method in Entropy-based Detection of DDoS attacks". In: doi: 10.2298/CSIS160905039P.

Mohaiminul Islam Guorong Chen, Shangzhu Jin (2019). "An Overview of Neural Network". In.

Murphy, Kevin P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press. isbn: 978-0262018029.

N. Murata, S. Yoshizawa and S. Amari (1993). "Learning curves, model selection and complexity of neural networks, in Advances in Neural Information Processing Systems 5". In.

OpenAI (2022). *Gym*. url: https://github.com/openai/gym. (accessed: 10.10.2022).

Peace, Green (2020). *18 animals that became extinct in the last century*. url: https://www.greenpeace.org.uk/news/18-animals-that-went-extinct-in-the-last-century/. (accessed: 08.01.2022).

Pelletier, Francis Jeffry (2000). "Petr Hájek. Metamathematics of fuzzy logic. Trends in logic, vol. 4. Kluwer Academic Publishers, Dordrecht, Boston, and London, 1998, viii + 297 pp." In.

Pharr, Matt (2005). *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*. Addison-Wesley Professiona. isbn: 978-0321335593.

Pramoditha, Rukshan (2021). *The Concept of Artificial Neurons (Perceptrons) in Neural Networks*. url: `https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc`. (accessed: 10.10.2022).

Robert Ornprapa P. Kumsap Chamnan, Suksuchano Sibsan (2019). "Modeling realistic 3D trees using materials from field survey for terrain analysis of tactical training center". In.

Segismundo S. Izquierdo, Luis R. Izquierdo (2018). "Mamdani Fuzzy Systems for Modelling andSimulation: A Critical Assessment". In: doi: `10.18564/jasss.3660`.

Simeone, Osvaldo (2018a). "A Brief Introduction to Machine Learning for Engineers". In.

– (2018b). "A Very Brief Introduction to Machine Learning with Applications to Communication Systems". In.

Spencer, Herbert (1864). *Principles of Biology*. Univ Pr of the Pacific. isbn: 978-0898757941.

Sung-Uk Choi Dongkyun Im, Seung Ki Kim (2018). "Physical Habitat Simulation with ANFIS Method". In.

Y. C. A. Padmanabha Reddy P. Viswanath, B. Eswara Reddy (2018). "Semi-supervised learning: a brief review". In.