Using deep reinforcement learning for exploring state spaces of finite state machines

Maciej A. Legas
2031545

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Master of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

December 15, 2021

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date 15/12/2021 .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date 15/12/2021 .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date 15/12/2021 .....

I would like to dedicate this work to my partner, Paulina, and all my wonderful friends. You know who you are.

Abstract

As the amount of knowledge amassed over time grows, so does the technological complexity [1]. With the rising technological complexity comes the rising advancement of the present-day research and development of technological devices. In our research, we review the areas of formal system modelling and reinforcement learning. Afterwards, we implement the use of deep reinforcement learning for the validation of finite state machines by creating a framework that allows an user to process their own finite state machine. We then evaluate the performance of chosen deep reinforcement learning algorithms on a number of implemented finite state machine environments and show how varying the performance of deep reinforcement learning algorithms can be, even with tuned hyperparameters. With the successful implementation of this project, we hope to push forwards the validation of models, via the use of deep reinforcement learning.

Acknowledgements

This thesis has become a reality due to the help and support of many people. First and foremost, I would like to express my gratitude to my mentor and supervisor, Dr Michael Edwards, for providing constant support and guidance, as well as pushing my ambitions with the project high, even when I had my darkest moments.

Furthermore, I would like to thank the members of my family, my partner, Miss Paulina Zawadzka, my course mates, and closest friends for being able to put up with my stressful behaviour during the course of this project. I would also like to individually thank my motivational mentor, Ms Philippa Jones, for straightening the curvy path towards the finish of this project.

Special thanks go to my friend from undergraduate studies, Mr Howard van Waard, for constant weekly motivational support, which was providing a real productivity boost.

Contents

List of Tables

List of Figures

.

Chapter 1

Introduction

Ever since the start of the Industrial Revolution in the 18th century [2], humanity started to experience a growing trend of the use of technology in their lives to ease the daily labour of an individual, especially with the concern of machine manufacturing [2]. After reaching the development of the first digital computers in the 20th century [3], we have reached a point where many of the electronic devices we use in our vicinity have software programmed microcomputers to automatically control the operation of the internal hardware of the device to improve or stabilize its performance. One of the best examples of this device usage growth is the evolution of the Internet after its introduction in the 20th century [4]. Taking into consideration the recent development of the Internet of Things that allows devices to share, process and analyse the data between each other to increase the overall user's experience [5, 6], at the time of writing, a standard U.S. household has 25 electronic devices with microcomputers connected to an Internet home network [7], with the devices being such as smartphones, TVs, and even washing machines. Of course, there are even more of such devices in households that do not have Internet connection capabilities.

Being pushed forwards with the goal of reducing the amount of needed manual labour, we have reached a point where currently, with the expansion of technological advancements, humanity is achieving things that before would have been seemed to be too abstract and unachievable with the past knowledge. One of the most interesting fields is the one of the development of artificial intelligence (AI), which ultimately aims to create computer programs that can solve problems and achieve goals on at least human-level performance [8], or in other cases would require human intelligence to solve [9]. Currently, the major usage of AI is via its subset of machine learning, which makes use of statistical models and algorithms, especially

neural networks, in order to search for patterns that allow a computer to solve tasks without needing to explicitly program how to solve the given task, usually via previously collected data [10]. Machine learning has been already successfully used to help with the process of increasing the quality of various areas of life, with one of the most renowned being the ongoing development of self-driving cars [11], lung cancer recognition [12], speech and facial recognition [13, 14], as well as the generation of such [15, 16]. It is clear to notice that this learning behaviour is quite comparable to how we, as humans, learn: while machine learning (ML) focuses on learning on the patterns of data, humans focus on learning via experience. Therefore, the concept of reinforcement learning (RL) has been also researched separately as its own subset of machine learning in order to develop an ability to learn in a real or artificial environment, without being dependent on existing data sets [17, p. 2-3]. One of the currently most known areas of reinforcement learning usage is the research of robot control, manipulation and navigation [18, p. 82].

With the mention of the evolution of technology and its growing popularity, it is important to notice how with the engineering knowledge that was gathered over time, the complexity of the produced appliances and devices has increased as well. However, what combines all of those successfully constructed devices, both those simple or complex in their mechanics, is the fact that most of them would most likely have not existed if not for proper planning, testing, and verification of their specifications, which with the later implementation of microcomputers in those devices started to involve software as well. As clearly indicated, this increasing internal complexity of devices has also increased the amount of work and time needed to design a specification for a working system. To help with this process in order to be able to ensure that the designed systems are logically correct while reducing the needed amount of prototyping or software testing, a number of techniques have been designed over time. The main ones on which this project will focus on are formal verification and finite state machines (FSMs), where formal verification allows us to verify the correctness of the implementation of a project, as well as its specification requirements [19, p. 65], and finite state machines allow us to abstractly model the workings of the given software and/or hardware in the form of a graph, with states a system can be currently in and transitions between those states [20, p. 48-50]. Finite state machines are declared using the laws of discrete mathematics, which allows us to verify the logical correctness of a given model using the previously explained formal verification.

As we have just mentioned, finite state machines are one of the ways to abstractly model the mechanics of a system to decrease the amount of manual labour needed for the planning

and testing phases of a project. In most cases, where an automatic solution has not been yet provided specifically for the given type of model, the formal verification and validation of such is still needed to be performed manually. While this can be a straightforward task when designing smaller models, in cases where a system reaches an amount of thousands of states or even more, the amount of time, thought, and test processing needed scales rapidly. If we take into consideration the possibility of using finite state machines for modelling a graph generated with the use of a mathematical algorithm to establish the possible states of a variable when exploring the field of, for example, complex analysis, the resulting state space might be extremely large. We can safely assume that manual model testing of it would be time inviable, although it is possible that even storing it in the random-access memory (RAM) of a standard home computer would not be possible in a case we wished to use an automatic formal verification method.

Therefore, we would like to provide a quality-of-life improvement to the process of software and hardware testing by incorporating the use of reinforcement learning into the process of exploring the state spaces of finite state machines for searching possible transition paths and catching any possible errors made while designing such FSMs. Ultimately, by this project we would like to pay a tribute to the ongoing development of technology.

1.1 Motivations

During our preliminary investigation, we have managed to establish that, at the time of writing, there is a possible existence of a research gap. While reinforcement learning has been found to be successfully used in previous projects for automated software testing [21], as well as being used to learn in environments based on a finite state machine [22, 23], we have not found any research that would base on creating a reinforcement learning framework that would learn how to traverse finite state machines for the further use of the training process of a RL model in the validation of a finite state machine. Therefore, this provides a motivation to construct and develop a solution that would take an insight into this gap. The existence of already developed graph visualisation and RL frameworks containing algorithm implementations push our confidence into the possibility of bridging this gap with the successful implementation of this project.

1.1.1 Objective

With the development of this project, we aim to provide a reinforcement learning based framework that would allow RL agents to traverse and explore the state space of finite state machines with their goal of increasing the distance from the start state. An additional focus is placed on allowing the processing of FSMs that normally would not be possible to be loaded and stored in RAM of a standard computer. This framework would let us use it to verify if the implemented design of a system validates pre-defined, specified criteria. To achieve this objective, a number of existing RL algorithms should be tested and evaluated for this task to compare their performance, and a rendering technique should be implemented to demonstrate the learning process of an agent. A successful completion of this project would like to be remarked by us as a possible stepping stone for pushing forwards the research in the fields of software and hardware testing.

1.2 Contributions

The main contributions delivered by this work can be seen as follows:

- **Thesis document explaining the project framework**

  A written document that is meant to help understand the research background and decisions made for the methodology and implementation of the included programming project.

- **Finite state machine environment for reinforcement learning**

  We provide a deep RL framework that is able to load a FSM, from a file that follows a defined template, and traverse through it to allow agents to learn to explore its state space.

- **Graph path rendering environment wrapper**

  A rendering wrapper that generates a heat map of the most amount of visits by the agents during their learning.

- **Implementation of a number of chosen FSMs**

  A number of chosen FSMs implemented as comma-separated values (CSV) files are provided for testing of the environment framework.

- **Evaluation of the performance of used RL algorithms on implemented FSMs**

  We review the learning performance of used RL algorithms, using their default hyperparameters, on each of the defined FSMs.

- **Using and evaluating auto hyperparameter tuning**

  Using auto hyperparameter tuning via Optuna [24], we repeat the training process using the tuned hyperparameters in the chosen RL algorithms and compare their performance against training with default hyperparameters.

1.3 Thesis overview

The following chapter 2 introduces and explains to the reader the concept of using finite state machines for system modelling and how they can be used for the formal verification of a system.

In chapter 3, we demonstrate the concept of reinforcement learning. This is accompanied by an explanation as of how does it differ from the other main fields of machine learning and how similar it is to the functioning of finite state machines. We also explain what does the concept of the *deep* keyword from the title of this research mean in the context of reinforcement learning and this project as a whole. Furthermore, we go through past research on using finite state machines in reinforcement learning.

Chapter 4 focuses on the frameworks and methods chosen to implement our reinforcement learning environment for finite state machines, what packages and projects did we decide to use, any issues we encountered, and how did we achieve to define the reinforcement learning part so it learns correctly how to pursue the exploration of the finite state machine.

In chapter 5, we discuss mainly the results of our experimentation with the RL algorithms and whether they were able to successfully learn and traverse the state spaces of the chosen finite state machines.

And lastly, in chapter 6, we summarize the main contributions and key points to take away from this project.

Chapter 2

Formal system modelling

In the planning phases of the creation of software and hardware, prototype high-level models are usually created to plan out the desired operation flow. However, in order to test the workings of such devices, some level of formality is required to verify their correctness. The main approach suggested by the field of theoretical computer science is the use of the concepts of automata theory.

Before we move onto defining what is the automata theory, we still need to clarify what automata (plural form of automaton) actually is. A dictionary definition cited from Merriam-Webster describes an automaton as *a machine or control mechanism designed to follow automatically a predetermined sequence of operations or respond to encoded instructions* [25], while Linz proposes a more restricting definition of *an abstract model of a digital computer* [26, p. 26]. Therefore, an automaton in the context of computer science and engineering can be described as an abstract state model that automatically responds to given input. If we apply this to automata theory, it can be simply understood as the study of automata, i.e. abstract machines [27, p. 1].

Using the definition alone, it should come easily understandable that the use of automata theory therefore allows us to use it as an approach to describe systems in a formal, modelled way. But how does the use of automata theory further help in the verification of a model? The explanation is in the usage of its core concepts of *language* and *grammar* [26, p. 17].

Let us assume the following set $\Sigma = \{a, b\}$ as the alphabet. Without the use of grammar, we can create any set of strings that contains at least one symbol of the defined alphabet to create a language L that bases on Σ. A language can be finite or infinite, depending on its set specification. However, in case we would like to use such a language formally, and study

whether a sentence using it is formed well, we need to consider the use of a grammar.

Definition 2.1 (Grammar) A grammar is usually defined as a quadtuple $G = (V, T, S, P)$, where [26, p. 21]:

- V is a finite, non-empty, set of non-terminal symbols, which can be otherwise named as auxiliary symbols as they are not obligatorily required as part of a sentence,
- T is a finite set of terminal symbols, which are the key parts of a sentence,
- S is a special symbol named the start symbol, which also has to be a subset of V, i.e. $S \in V$,
- P is a finite, non-empty, set of production rules.

Let us define an example grammar $G = (V, T, S, P)$, where:

$$V = \{S\}$$
$$T = \{a, b\}$$
$$S = \{S\}$$
$$P = \{S \to aSbS, S \to bSaS, S \to \lambda\}$$

If we apply the grammar G to the language L, with the lambda symbol defining an empty string, we obtain the following possible example outputted sentences: $L(G) = \{ab, ba, abab, baba, ...\}$. By combining the use of the concepts of formal language and grammar, we can easily rule out grammatically incorrect sentences, such as $\{aa, bb\}$, and mark them as invalid, henceforth providing what is known as a *regular language*. Furthermore, as these concepts allow for the performing of formal verification of symbols, these concepts have been involved into the further subfield of automata theory, which is the modelling of finite state machines.

2.1 Finite state machines

Finite state machines, also known as finite automata, are a mathematical model of computation and come as a product of automata theory [27, p. 5, p. 37]. The *finite* keyword means there is a limited, finite amount of states, therefore finite state machines with a formal description allow us to perform verification on them. Although they are usually used to provide a simple model

of a computing device [20, p. 48], that does not mean that we are limited with its modelling capability to only electronic systems, as we can describe any automaton that can be traversed using an input string.

Definition 2.2 (Finite Automata) Finite automata are usually defined as a quintuple $FA = (Q, \Sigma, \delta, q_0, F)$, where [20, p. 49]:

- Q is a finite set of states,
- Σ is a finite set of input symbols, also known as the alphabet,
- δ is a transition function that returns a state, being given a state and an input symbol as arguments. The transition function can also be seen as a function that maps $Q \times \Sigma \rightarrow Q$,
- q_0 is a start state, which also has to be a member of the set Q, i.e. $q_0 \in Q$,
- F is a set of final states, which is a subset of Q, i.e. $F \subseteq Q$. This set can be empty in some cases.

Let us consider the following example of a light switch for further explanation. We start at an "off" state. Pushing the switch will turn the light on, and therefore move us to the state "on". If we push the switch again, we turn off the light, and go back to the state "off". Now, we would like to formally describe its mechanics in such a way that is easily reproducible and verifiable, so that even if the outside design varies completely from one light switch to another, the base workings of them remain the same. This is one of the examples where modelling via the concept of finite state machines can help with such a task.

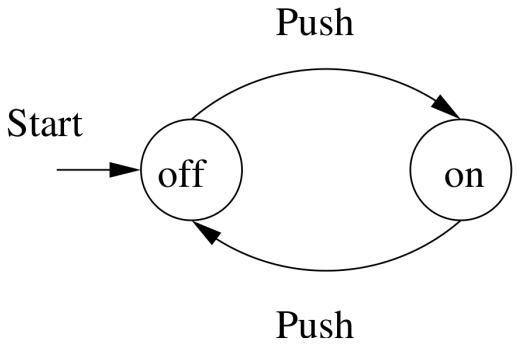


Figure 2.1: A state transition diagram of a finite state machine of a light switch [27, p. 3].

If we look at the graph above, we can deconstruct the finite state machine into each of its elements:

$$Q = \{off, on\},$$
$$\Sigma = \{Push\},$$
$$q_0 = \{off\},$$
$$F = \emptyset.$$

We need to also include the transition function δ, which is given by:

$$\delta(off, Push) = on,$$
$$\delta(on, Push) = off.$$

However, in cases of more sophisticated finite automata, a clearer way of visualizing the transition function can be achieved via the use of a *state-transition table*:

Table 2.1: A state-transition table of a light switch.

Input	Current state	Next state
Push	off	on
Push	on	off

With those elements explicitly described, we can easily (re)construct the light switch mechanic.

2.1.1 Deterministic Finite Automata

Opportunistically, the provided finite state machine that was explained in the previous section is also of the type that this project will focus on, which is deterministic finite automata (DFA).

Definition 2.3 (Deterministic Finite Automata) A deterministic finite automata is usually defined as a quintuple $DFA = (Q, \Sigma, \delta, q_0, F)$, where each of its symbols has the same function as a standard finite automata [20, p. 53].

Besides deterministic finite automata, finite state machines can also be divided into the type of non-deterministic finite automata [20, p. 53-54]. The key difference to remember between those two types is that the cardinality of the transitional function Q in DFA is equal to 1, i.e.

$|Q| = 1$ [20, p. 53]. This means that in the case of deterministic finite automata, one input from a given state can lead to only one state, whereas in the case of non-deterministic finite automata (NFA), one input from a given state can lead to multiple states, which is a clear trouble sign for the validation of such a model. Fortunately, NFA can be converted to DFA by a number of techniques, with one of the standard ones being the powerset construction method [20, p. 55].

The image below shows another example of a deterministic finite automaton for clarity, with a final state F.

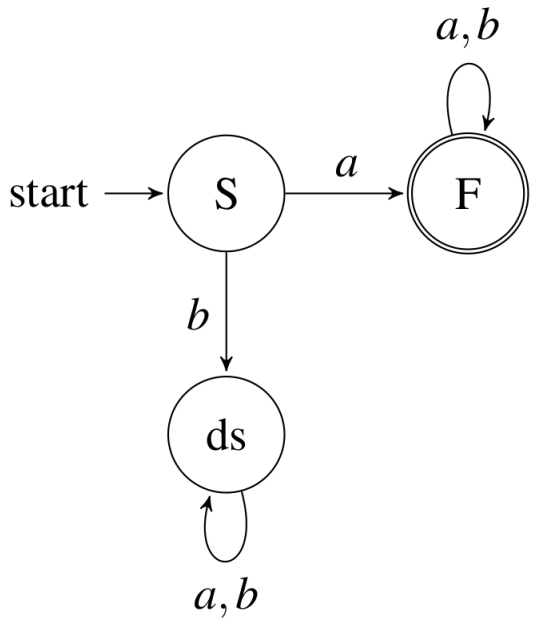


Figure 2.2: A state transition diagram of an example abstract deterministic finite automaton [28, p. 59].

2.2 Model checking and formal verification

As we have shown in the previous section, we can abstractly model discrete systems to describe their operation. The determinism of model types such as DFA is what allows us to further test such models via formal verification. As Seshia et al. [29, p. 75] describe, modelling of a system is the initial, and often the most crucial step in the verification of such a system. Grout describes formal verification as *identifying the correctness of hardware and software design operation* by the use of formal mathematical proofs [19, p. 65], which in the case of system models is defined as model checking. What is important to note is that the captured, modelled systems usually do not cover all of the system steps, but rather focus on the steps that are compulsory

for its functioning. This is due to the rather simple reason of keeping the model size low and easy to understand [29, p. 155]. Still, a huge model size usually does not particularly make a problem for the model checking testing part of a project, as most mathematical proofs that use temporal logic [29, p. 27] and proof by induction [27, p. 19] overcome this obstruction just by verifying the correctness of the used grammar in a model.

However, while we can check for the correctness of pretty much any deterministic models mathematically, formal verification alone does not contain for a heuristic check of a model. What we mean by this is that it does not perform validation, which is defined by Grout as examining the behaviour of a system in order to validate the correctness of its operation, whether by simulation or prototype evaluation [19, p. 65]. While there are automatic methods that combine validation and verification, such as hardware description languages [19, p. 65], they still highly depend on the reduction of the amount of discrete states in the state space for the further testing, which can remove auxiliary states that an user may have deemed as important.

Taking this point into consideration, in this research we would like to therefore base the notion of testing of finite state machines, or rather, deterministic finite automata, on the exploration of their state spaces for their validation. While there is a process in machine learning named grammar induction [30, 31], it is mainly used to train machine learning models with neural networks on data sets for them to produce correct input strings for language modelling and generation. Therefore, with the main focus of exploration in mind, we would like to shift our focus to the use of reinforcement learning, which will be explained in the following chapter 3.

Chapter 3

Reinforcement learning

Reinforcement learning has its core origins in the biological concepts of *experiental learning*, *reinforcement* and *reward system*. We, as humans, learn throughout our lives mainly by experience, whether that be for education, work, personal development, or just basic functioning as a human being [32, p. 1-8]. In cases where we are not forced to experience a situation, and there is no intrinsic motivation, i.e. immediate *reward* that can stimulate our pleasure systems in our brain by experiencing it [33], if we wish to experience it anyway, we usually aim to push our motivation via extrinsic rewards. Let us take achieving a successful completion of an academic degree as an example. While the learning and assignment processes may not be particularly pleasurable on the academic path, students usually *reinforce* their behaviour of constant learning by having the extrinsic reward of achieving higher education in the future in mind. Of course, those that find their studies intrinsically rewarding will most likely be more motivated than those that can solely base on the extrinsic rewards. Having said that, it is also important to remember about the existence of negative reinforcement. If we experience some of the actions have a personal result to us in either intrinsic or extrinsic negative rewards, we will be less motivated to perform them later. Lastly, both of these rewards together form the *reward system*.

These concepts have a direct use in reinforcement learning, which can be classified as one of the main areas of machine learning, as shown on the Figure 4.4 below.

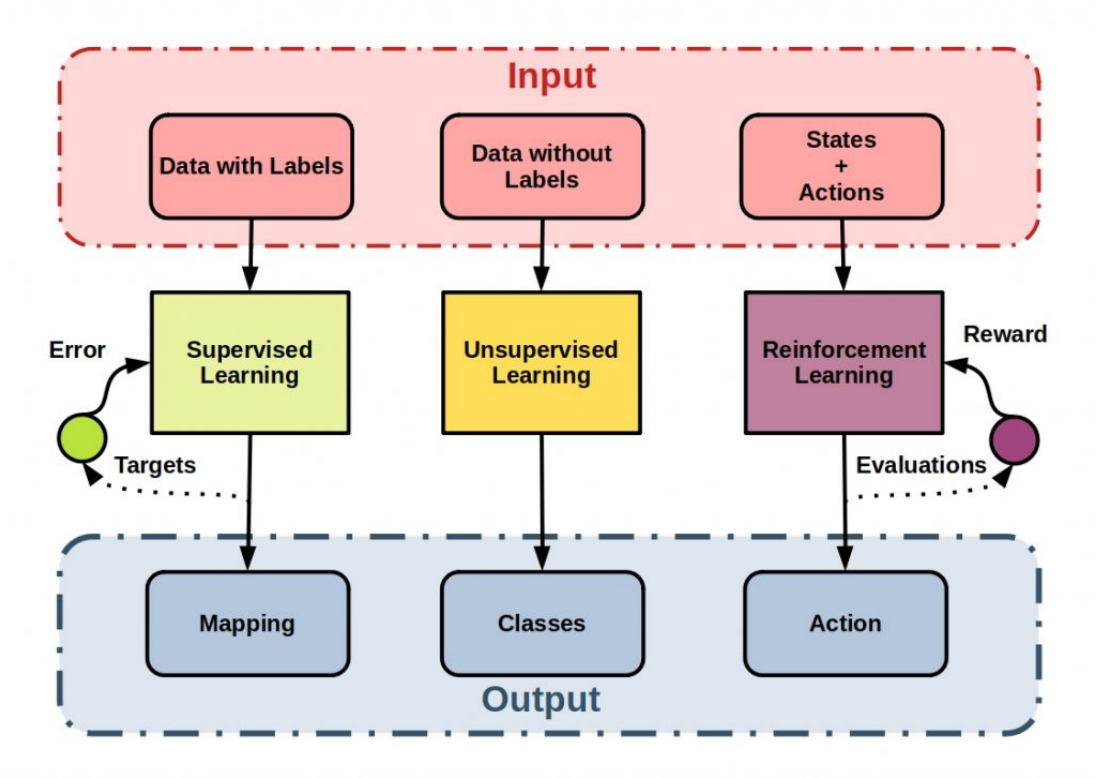


Figure 3.1: Overview of the main areas of machine learning [34].

Correspondingly, while we can see that reinforcement learning does seem similar in its generalized view to supervised learning, the main difference comes from the fact that supervised learning bases on learning on data that is already collected and labelled by provided data sets, while reinforcement learning focuses on learning on data that has to be collected sequentially in a specified environment, and therefore learn via interaction and eventually, experience.

To formalize the environment that is hence used in reinforcement learning, we use what is named Markov Decision Processes (MDPs).

Definition 3.1 (Markov Decision Process) A Markov Decision Process is a quintuple (S, A, T, R, γ), where [17, p. 17]:

- S is the state space,
- A is the action space,
- $T : S \times A \times S \to [0, 1]$ is the transition function, represented by a set of conditional transition probabilities between states, i.e. $T(s'|s, a)$ represents the probability that taking the

action a while being in state s will lead to state s' with the next time step,

- $R : S \times A \times S \to \mathscr{R}$ is the reward function, where $\mathscr{R}$ is a continuous set of possible rewards, i.e. $R(s'|s,a)$ represents the reward for taking the transition to state s' when being in state s and taking action a,
- $\gamma \in [0,1)$ is the discount factor, which is usually used to discount rewards based on how far do they appear in the future.

In order to interact and learn via the use of a MDP model, reinforcement learning makes use of an agent as the main learner and decision maker [35, p. 47-48]. To further demonstrate this process, we will examine Figure 3.2 below. An agent starts in a start state S_0 and therefore receives an initial observation of S_0. By performing an action A_0, the agent receives a reward R_1 and a new state observation S_1. With this ongoing sequence, i.e. S_{t+1}, R_{t+1}, the goal of the agent is to maximize the rewards provided by the environment, and therefore (approximately) learn the probabilities of the transition function T with the use of the reward function R (and possibly the discount factor γ).

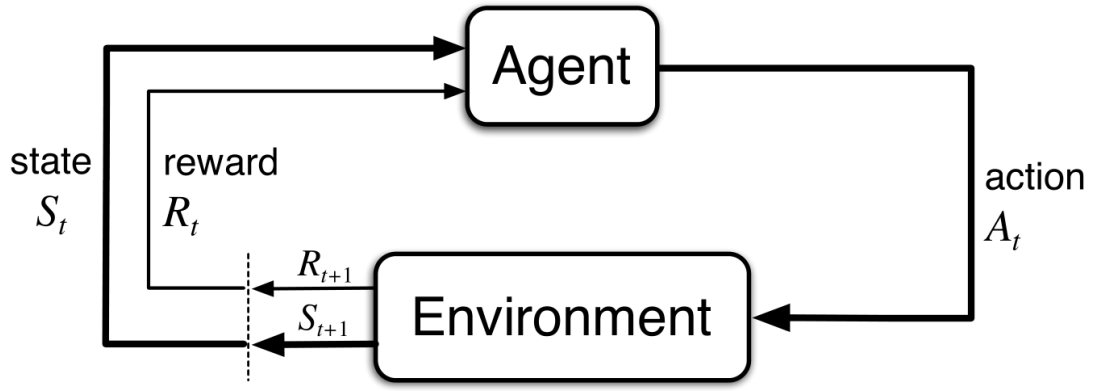


Figure 3.2: A representation of the agent-environment interaction in reinforcement learning [35, p. 48].

To formalize this further, an agent aims to maximize the expected *discounted return* gained from completing an *episode* in a given environment by choosing actions A_t [35, p. 55]:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{3.1}$$

where γ as a parameter, as explained in the definition 3.1 of MDPs, allows us to discount the value of future rewards. The use of the discount factor has a simple explanation: the lower

it is set, the more agent focuses on receiving maximum immediate rewards, and vice versa - the higher it is set, the agent focuses more on the possibility of receiving higher rewards in the future.

To decide what actions to choose to receive such rewards, the agent follows the actions of a *policy* π, which maps the probabilities of choosing action a at state s [35, p. 58]:

$$\pi(a|s) = T[A_t = a|S_t = s]. \tag{3.2}$$

The performance of a policy is established via the use of a *value function* $v_\pi(s)$ [35, p. 58]:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s], \tag{3.3}$$

where $\mathbb{E}_\pi$ is a prediction of the value of the expected return while being in state s and following policy π afterwards among the time step t. Another function used is the *action-value function* denoted $q_\pi(s,a)$ [35, p. 58]:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s, A_t = a], \tag{3.4}$$

where $\mathbb{E}_\pi$ is a prediction of the value of the expected return while being in state s, but also taking the action a, and then following policy π among the time step t. Therefore, *optimal policies* have the following value and action-value functions defined [35, p. 62-63]:

$$v_*(s) \doteq \max_\pi v_\pi(s), \tag{3.5}$$

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a). \tag{3.6}$$

There is however an important property of the value function, which is named the *Bellman's equation* [35, p. 59]:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s], \tag{3.7}$$

where this condition ensures that the value of a state s is equal to the immediate, discounted value of the expected next state alongside the expected reward on the next time step t.

With the Bellman's equation explained, we can also define the *advantage* function here:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s), \tag{3.8}$$

which by subtracting the value function from the action-value function provides a description of how well the action a performed in contrast to the expected return without the action [17, p. 20]. This advantage function is often used to evaluate the learning process in deep reinforcement learning algorithms.

It is important to remember that we can only model environments with the MDP model architecture in mind, if the transition function follows what is called the *Markov property* [17, p. 17-18]:

$$T = [S_{t+1}|S_t] = T[S_{t+1}|S_1, ..., S_t], \tag{3.9}$$

where this property essentially ensures that the probability of a future next state S_{t+1} can be only dependent on the current state S_t, and guarantees the determinism of the model.

With the *Markov property* and *Bellman's equation* in mind, if we recall the Definition 2.2 of the finite automata model architecture, we can see a few clear similarities between the MDP and DFA architectures. This is particularly visible if we consider a deterministic finite state machine modelled as an environment, which would therefore be a fully observable Markov Decision Process, and furthermore shows the potential of using reinforcement learning in this research.

With the foundations of reinforcement learning explained in this section, we would like to move onto describing methods that facilitate the learning mechanic and derive optimal policies. In order to so, we need to first return our focus to the main objective of this research, which is the exploration of potentially large state spaces of finite state machines. While there is a number of existing traditional reinforcement learning methods, such as Q-learning [35, p. 131], dynamic programming [35, p. 73] and Monte Carlo methods [35, p. 91], the main issue with using these methods is that they cannot handle high-dimensional MDP state spaces s on their own, as they are too complex for their mechanics. Furthermore, even if that was possible and if we consider that we know the MDP dynamics of the environments and therefore these methods should be sufficient, the problem arises in the fact that they store each probability of the best evaluated policy π. This would turn out to be infeasible in our project, as we wish to be able to use the possibility of approximating the probabilities in order not to hit the memory constraint in case of extremely large state spaces. Fortunately, with the recent discoveries in the field of deep reinforcement learning, we are able to do so.

3.1 Deep reinforcement learning

Currently, the main research focus in the area of reinforcement learning is based on the usage of deep learning. As shown on Figure 3.3 below, deep learning has its part in all of the main three areas of machine learning, including not only reinforcement learning, but also supervised and unsupervised learning.

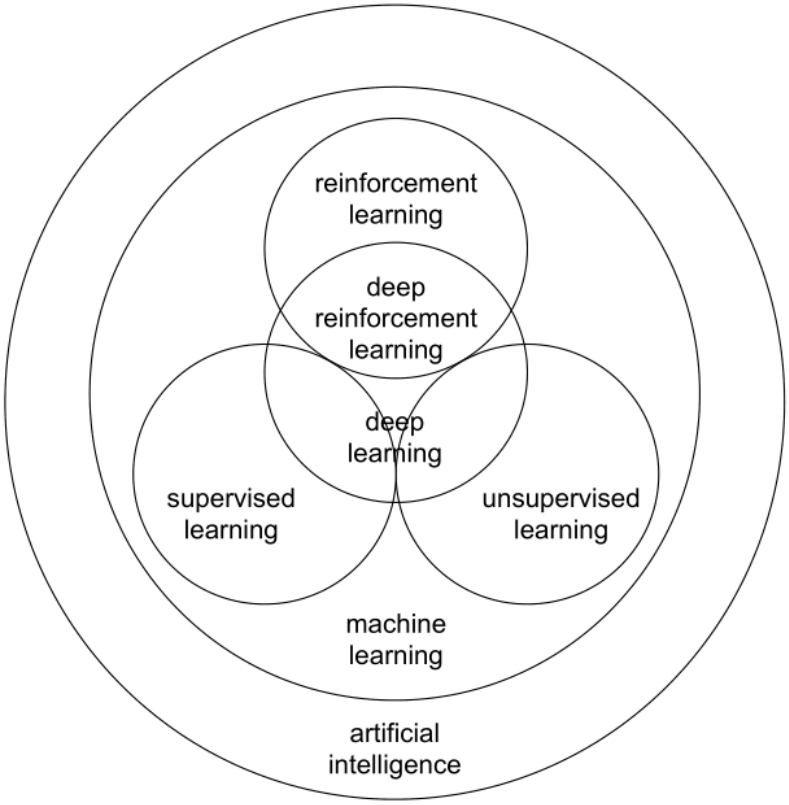


Figure 3.3: The relationship between deep learning and other areas of machine learning [18, p. 9].

Deep learning comes as a natural extension to machine learning and focuses on the use of neural networks, which can be otherwise described as function approximators [36, 37]. A neural network consists of multiple perceptrons, which base on received inputs and their current weights to further process it with a transfer function and decide whether to activate or not, i.e. provide output, using their specified activation function, as shown on Figure 3.4 below. The weights of these perceptrons are usually randomized during their initialization.

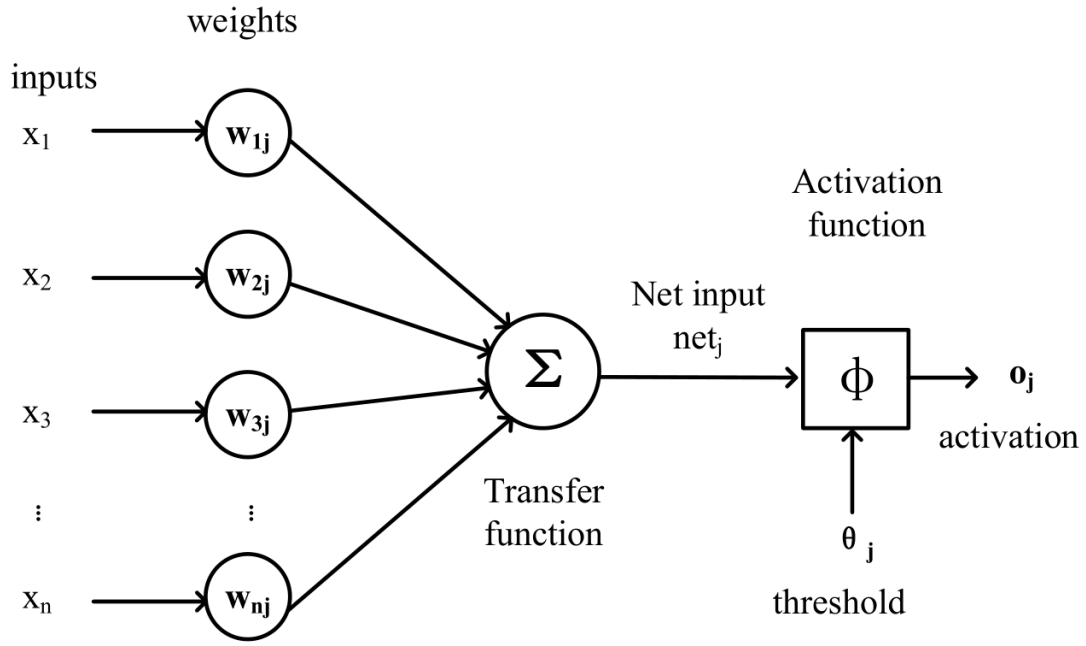


Figure 3.4: An image of a perceptron/artificial neuron [36, p. 587].

This behaviour has found its main use in the tasks of classification and regression in the field of supervised learning [38, p. 107]. Such neural networks are usually defined to learn via the use of backpropagation and mean squared error (MSE) [38, p. 129],

$$MSE(\hat{F}) = \int p(\hat{F},F)(\hat{F}-F)^2 d\hat{F}dF, \tag{3.10}$$

where F is a given variable, for example, a class of birds to identify, or more related to our case, *reward*, and $\hat{F}$ is an estimate of a sample of F. MSE often takes the role of a loss function, showing how well the neural network performs, and therefore allowing the perceptrons to adjust their weights correspondingly to the mean squared error via the backpropagation algorithm.

Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), differ from the standard machine learning mechanics by their architecture. In deep learning, models of neural networks consist of an input layer, one or more hidden layers, and an output layer, as shown on Figure 3.5 below. The ability of combining multiple artificial neurons with multiple layers allows us to approximate more complex tasks that a stan-

dard "shallow" neural network would not be able to handle, however with the cost of possible learning instability and divergence [18, p. 21].

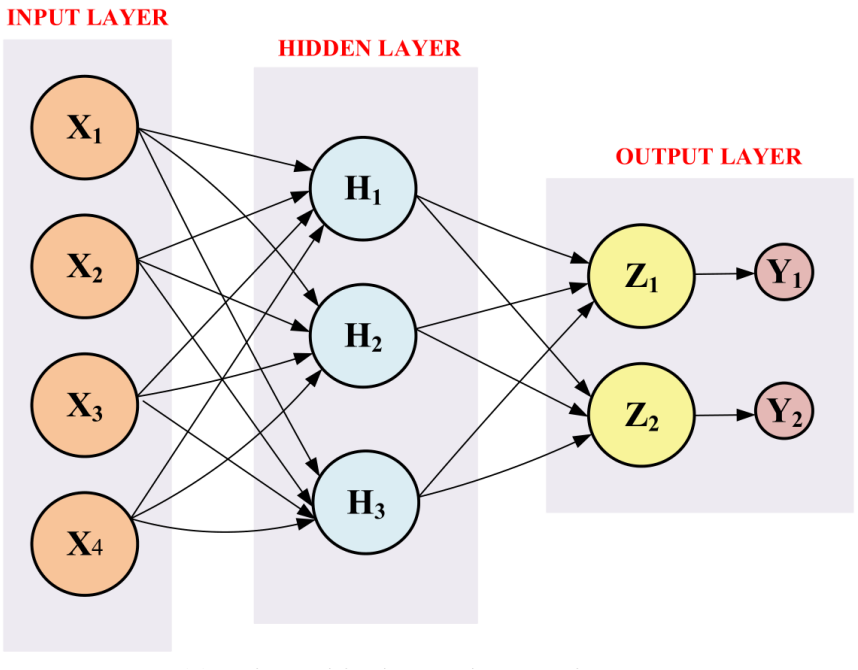


Figure 3.5: An example deep neural network [36, p. 587].

In deep reinforcement learning, we introduce the parameter θ, which represents the weights of the perceptrons in deep neural networks [18, p. 21]. By the use of this parameter, we can approximate any of the previously established reinforcement learning functions with the use of deep neural networks if needed, that is:

$$\hat{v}_\theta(s), \tag{3.11}$$

$$\hat{q}_\theta(s,a), \tag{3.12}$$

$$\pi_\theta(a|s). \tag{3.13}$$

An example of an agent following an approximated policy $\pi_\theta(s,a)$ is shown on Figure 3.6 below.

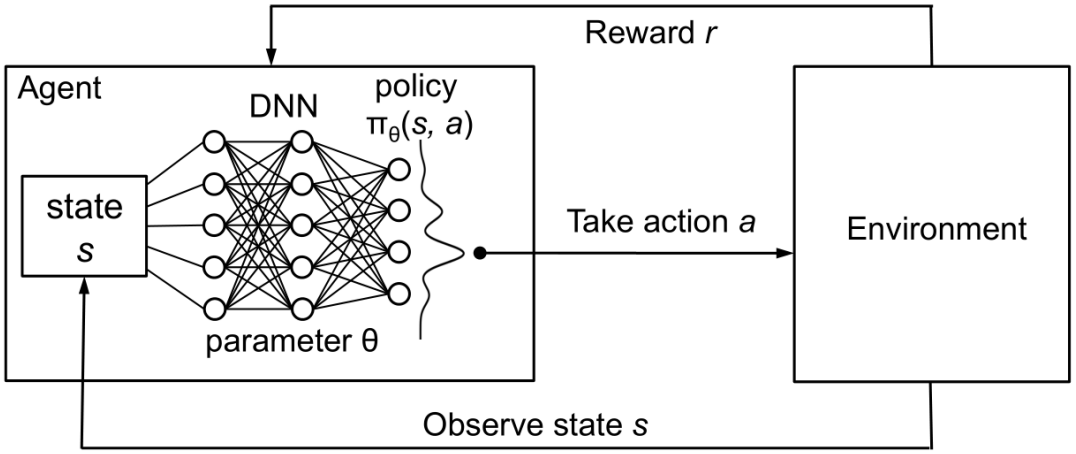


Figure 3.6: How deep learning is used in deep reinforcement learning [39, p. 51].

With the introduction of deep learning, it should come as no surprise that there has been an extensive amount of research in reinforcement learning regarding the possible ways of approaching the problem of maximizing the cumulative reward, i.e. $v_*(s) \doteq \max_\pi v_\pi(s)$. There is a numerous amount of algorithms available, which further divide into two sub-classes [40]:

- Model-based: An agent has access to, or learns, a model of the environment, that allows them to predict what happens if it makes an action a at state s, using a function $Model(s,a)$. They are usually more sample efficient than model-free approaches [40,41].

- Model-free: An agent does not have access or does not learn the model of the environment, and therefore focuses solely on trying to optimize the policy. Due to this, they are less sample efficient than model-based approaches.

- On-policy: An agent uses and updates a policy $\pi_\theta(a|s)$ using an objective function $J(\pi_\theta)$. The observation data from the environment for further updates of the policy gets collected via the most recent version of the policy $\pi_\theta(a|s)$, which is also how the agent explores. Policy algorithms often include the use of an additional value function approximation, i.e. $V_\phi(s)$ for $V_\pi(s)$.

- Off-policy: Algorithms of this kind usually base on an objective function similar to the *Bellman's equation* (Equation 3.7), where an agent learns approximate action-value functions $Q_\theta(s,a)$ to find an optimal action-value function $Q^*(s,a)$ for the policy $\pi^*(a|s)$. The off-policy part of this kind of algorithms means that the observation data

collected from an environment comes from the optimization of the action-value function, and hence does not solely depend on the exploration steps of the policy.

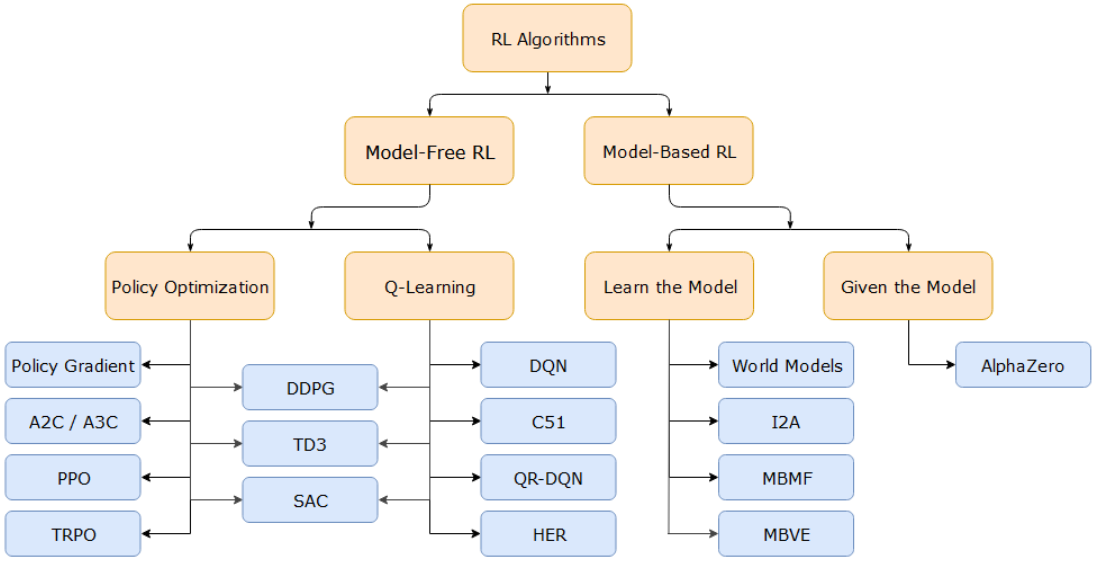


Figure 3.7: A non-exhaustive taxonomy of reinforcement learning algorithms [40].

Due to the further methodology constraints of the project, such as the use of multiprocessing, this section will now focus on explaining policy gradient methods, in particular the Proximal Policy Optimization (PPO) [42] and Advantage Actor Critic (A2C) [43] algorithms visible on the Figure 3.7 above. These algorithms are model-free, off-policy deep reinforcement learning methods that have been successfully used in many other researches [44, 45].

3.2 Policy gradient methods

The policy gradient methods base on using the gradient of an objective $J(\pi_\theta)$ to maximize the cumulative expected reward, which is [39, p. 51]:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t r_t] = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{3.14}$$

A basic policy gradient method focus on the optimization of a policy via performing gradient descent on the policy parameters θ [39, p. 52]:

$$\theta \leftarrow \theta + \alpha \sum_{t} \nabla_{\theta} log \pi_{\theta}(s_t, a_t) v_t, \tag{3.15}$$

where α is a set *learning rate*,

$$\alpha = [0, 1]. \tag{3.16}$$

The function of the learning rate, also known as *step size*, is to adjust as to how much do we change the parameters θ on each update of the policy $\pi_{\theta}(a|s)$. Higher values of α might bring convergence faster, but can get stuck on a local maximum of the policy gradient, therefore not being close to a global maximum. Lower values of α will take more time steps to converge, but do not have the possible limitation of higher values that we have just noted [38, p. 94].

3.2.1 Advantage Actor-Critic

The advantage actor-critic (A2C) algorithm works by having two deep neural networks - one for the actor, which is the policy:

$$\pi(a_t|s_t;\theta) \tag{3.17}$$

and another for the critic, which is the state-value function:

$$V(s_t;\theta_v) \tag{3.18}$$

The actor performs actions, and the results of these actions alongside them are processed by the critic, which provides feedback back for the actor. After a number of time steps, both the actor and critic gets updated using an estimation of the advantage function (defined in Equation 3.8) [43, p. 4]:

$$\nabla_{\theta'} log \pi(a_t|s_t;\theta') A(s_t, a_t; \theta, \theta_v), \tag{3.19}$$

which is an advantage over most of the other on-policy RL algorithms that have to wait until an episode has ended to update their policies.

The following pseudocode for asynchronous advantage actor critic (A3C) explanation comes from the work of Mnih et al. [43, p. 14], and is shown below as Figure 3.8:

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 3.8: Pseudocode of Asynchronous Advantage Actor-Critic [43, p. 14].

A2C, just as A3C, uses multiple workers to benefit from the amount of received data observations from the multiple interactions in an environment. The difference between the workings of A2C and A3C is solely in the thread synchronization - it has been found that waiting for the workers does not bring any efficiency issues, and the synchronous, deterministic (A2C) version is actually more cost-efficient than the asynchronous (A3C) version [46].

3.2.2 Proximal Policy Optimization

The proximal policy optimization (PPO) algorithm takes the ideas of multiple workers and the actor-critic networks from A2C, and combines them with an use of a trust region, which is a solution taken from a previously developed Trust Region Policy Optimization algorithm [47, 48].

The main idea behind the implementation that we will be using of the PPO algorithm, which is named PPO-Clip, is that to ensure an approximate global maximum is found, the algorithm clips stochastic gradient updates to the policy parameters θ to ensure that any new policy is not too far off from the previous one [49]. This limits the effect that the advantage function $\hat{A}_t$ has on new policies, which although can extend the learning process, usually results in a safer learning convergence, as the new policies will not be affected as much by sudden large

negative/positive rewards [42].

The general pseudocode following pseudocode explanation comes from the work of OpenAI [49], and is shown below as Figure 3.9:

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, ...$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Figure 3.9: Pseudocode of Proximal Policy Optimization [49].

3.2.3 Exploration vs exploitation

During the course of the previous section, we have mentioned the term *exploration* a couple of times. The discussion between balancing exploration and exploitation has been well documented in literature, and is one of the fundamental problems of reinforcement learning [17, p. 66-67]. One of the standard and most known methods in traditional reinforcement learning is named ε-greedy. In this approach, an agent selects the current best (most rewarding) action for state s in $Q(s,a)$ with probability $1-\varepsilon$, and selects a random action with the probability ε, where $\varepsilon \in (0,1)$ [18, p. 14]. In the case of deep reinforcement learning, this problem is mostly mitigated by the constant change of new policies which has an initial exploration effect on the value function. However, that tends to be not enough, and therefore an *entropy coefficient* is often used. The entropy coefficient, often denoted with the symbol H or S, is generally used in the gradient function of θ' to give more cost to rewards received by actions that the policy

is currently dominantly using [50]. This is denoted as *entropy regularization*, as it helps to regularize policies to "push" them from any local optimum.

3.3 Usage of deep RL with DFA

In the beginning of this section 3, we mentioned the similarity of deterministic finite automata to Markov decision processes. While we were not able to find any research in literature directly regarding the traversal of large state spaces of finite state machines, we have managed to find works that are relevant to this project. The main key part that we have established is that deep reinforcement learning has been successfully used in the past for learning deterministic finite automata.

Perhaps the most relevant work has been shown in the research of Lloyd-Roberts [22]. In their research, they aimed to support the formal verification of railway interlocking mechanisms, which is similar to our main aim. For this objective, they modelled a pelican crossing as a finite state machine, and further used it as an environment for a reinforcement learning agent. The goal of using reinforcement learning was to find an optimal path between all of the states in the state space. The reward shaping focused on providing positive rewards for newly discovered states, and negative rewards for states that have been already discovered, with harsher rewards for repeatedly being in the same states. A final large positive reward was provided for finding all of the transitions and states in the state space. Using this environment, they trained an agent using an algorithm named Deep Q-Learning (DQN) [17, p. 27], which managed to find an optimal path throughout the pelican crossing, and therefore further support the formal verification of the finite state machine.

Another work that we would like to mention is the research of Hasanbeig et al. [23] of DeepSynth, in which they defined an algorithm for training deep reinforcement learning agents, where the reward system is sparse and non-Markovian. For their research, they based on training on the environment of the Atari game *Montezuma's Revenge*, which requires a set sequence of inputs to receive an extrinsic reward. Therefore, to make the problem more understandable for an agent, the visual input has been divided into labels of visible objects on-screen, and then provided as a synthesised deterministic finite state machine to the agent. Intrinsic reward was provided if the agent interacted with a new object, i.e. label, which can be further described in our understanding as a state in the state space of the deterministic finite state machine. This framework has been found to be successful in their evaluation, and furthermore, managed to explore and learn the optimal actions (path) of the *Montezuma's Revenge* environment.

Finally, Dulac-Arnold et al. [51] in their study of deep reinforcement learning in large discrete action spaces have approached the issue of environments with extremely large action spaces. Their research focused on discretizing the MuJoCo [52] physics simulator and creating a framework that would allow to perform an approximate nearest-neighbours search for closest discrete actions. While their solution of the *Wolpertinger architecture* is excessive for this project, as it was created with the goal of approximating 1 million possible discrete actions for environments with 20 possible time steps in their research, it stands as a solid background for learning MDPs with large state spaces.

Chapter 4

Methodology and implementation

In this section, we explain the methods and solutions we used for the implementation of this project. We also document any issues that arose during implementation and the ways we resolved them.

4.1 Programming language

For the implementation of this project, we based on using the *Python* programming language [53]. The major advantage of choosing Python is the fact there is an abundance of data science and machine learning packages developed for this language, which easens the amount of implementation work needed. Furthermore, for clear code documentation and easy interaction with output, the *Jupyter Notebook* application [54] is used, which allows for a clear division of code and documentation via the use of cells. This also allows to run individual parts of code, rather than having to unnecessarily divide code into their own functions or files.

4.2 Integrated development environment

To implement the code required for this project, we used *Visual Studio Code* [55] as our integrated development environment (IDE). This is due to a similar reason as to the use of Python, as Visual Studio Code contains a lot of extensions, especially for popular programming languages such as Python. Moreover, Visual Studio Code allows for an easy integration with Jupyter Notebooks, making it possible to run the notebooks inside Visual Studio Code.

4.3 Virtual Python environment

To collect together all of the Python packages that we have used during the implementation, we used *Anaconda* [56], which allows to create *conda* environments that isolate the installed Python packages from other projects. A major advantage of this is that it allows to list all of the packages used in the project in a separate list, and further allow for an easy creation of a reproducible conda environment.

A readme file is attached with the project that explains how to create a new conda environment with all of the installed Python packages of this project. It is also included as Appendix B.

4.4 Finite state machine file structure

To define the finite state machines in a file structure, we decided on using the comma-separated values file type (CSV) as they are easily both readable by programs and programming packages, and human understandable. The following columns were declared (underscores removed for reading clarity), where an individual row represented one state:

- Unique ID: An unique ID for the state, which allows to visualize the path of the agent. An important property is that the Unique ID value must be above 0, i.e. $ID \in \mathbb{N}^+$.
- State name: A state name of the state.
- Start state: Whether the state is a start state, i.e. $s \in q_0$.
- Possible discrete actions (transitions): The amount of possible transitions that can be taken from the given state.
- Transition names: The label of each transition.
- Transitions to states: Provides information about which states do the given transitions lead to.
- Discretized state name: A converted state name (as the agent cannot read strings) to a discrete number. This allows us to optimize the environment in cases where states have the same transitions leading to them and from them, i.e. states that have the exact function have the same discretized state name.

4.5 Agent environment

In order to define the agent environment, we used the *Gym* framework from OpenAI [57], which besides providing a number of already implemented environments to train RL algorithms on, allows for the creation of custom environments. The further advantage of creating an environment for the agent via Gym is that its use is well supported by a number of RL algorithm implementations.

4.5.1 Start states

On the initialization of the environment, if no preset start state has been defined, our framework will scan the file for any start states. If there is only one start state declared in the CSV file, the framework will use it for the further training of any agent. In case that multiple start states are found, the framework will ask the user to either set one start state or randomize the start state on each reset of the environment. The possibility of using multiple start states is a positive benefit of our environment, as it allows for what is essentially training the agent on "multiple" deterministic finite automata in the same environment.

4.5.2 Memory constraint

To ensure that we do not store the entire finite state machine in memory, each time the agent makes an interaction with the environment, we open the provided CSV file with the finite state machine and scan it row by row. For example, if we make a transition to another state, our framework will first scan the file for the state the agent is currently in, and then scan it again in search for the state the agent decided to transition to. While the time complexity will scale with larger files rather quickly, it is one of the only possible ways other than using chunks of the file to deal with the memory limitation.

4.5.3 Using knowledge about the finite state machine

While we would most likely not consider this technique as *model-based*, we provide an ability to scan the CSV file for the observation space limits, such as the maximum amount of states. This by itself should have a boosting effect to the convergence speed of a model. If wished otherwise, it is also possible to pass a *False* argument to not scan the file and use default observation space limits, which may result in unstable and/or slow learning when used on small state spaces. We also provide an integer argument to the framework that allows to use

the default observation space limits after scanning a set amount of rows, therefore breaking the scan.

4.5.4 Reward shaping

The shaping of the reward function has been thought of by having in mind the exploration of the state space. Therefore, at each step, if an agent encounters a state that he has not been in yet during an episode, he receives a reward that is equal to the amount of states he has visited during the episode, not counting the start state. For example, an agent that traverses through 10 states from the start state, upon traversing to s_{10} will receive a reward of:

$$reward(s_{10}) = 1+2+3+4+5+6+7+8+9+10 = 55. \tag{4.1}$$

However, if an agent "visits" a state that he has already traversed to during the episode, it will receive a reward of 0, and the episode will be terminated, i.e. considered as done. A negative reward was considered, although it was decided that it might negate the transition probabilities too much. We terminate the episode to ensure that the agent does not stay in the same state in case of getting trapped in a "dead-end".

This termination of episodes also means that we omit the use of any terminal states in our environment framework, as we only focus on finding most of the state in the finite state machine.

4.5.5 Action space

A variable action space was decided to be used, which means that at each step of the environment, the agent is provided a discrete action space equal to the amount of transitions that can be taken from the current state. The chosen discrete action is then used to choose from a state to transition to from the *Transitions to states* field.

4.5.6 Observation space

To combine the multiple observations that we wanted to provide to the agent, we used the *Dict* gym observation space [58] with the following observations (underscores removed for reading clarity):

- Current state: The current discretized state the agent is in.

- Past states: The history of the states the agent traversed through, including the current one.
- Transitions to states: To which discretized states the agent can currently traverse.
- Amount of states visited: How many states the agent has "visited" already.

Each of the observations was declared as an *integer* type. The observations use a *Box* gym observation space, which means that they are declared as closed intervals [59], which can be shaped as vectors of n dimensions. Therefore, the limits of each observation were dependent on whether the environment scanned the provided CSV file for maximum values, except for the low bound, which is always set to 0.

If the file was scanned, the following observation space limits were provided:

- Current state:
  - High bound = Amount of states in the file + 1,
  - Shape = 1-dimensional
- Past states:
  - High bound = Maximum value a 64-bit integer can hold,
  - Shape = Amount of states in the file + 1
- Transitions to states
  - High bound = Maximum value a 64-bit integer can hold,
  - Shape = The maximum amount of transitions found from a state in the file
- Amount of states visited
  - High bound = Amount of states in the file
  - Shape = 1-dimensional

The addition of 1 to the high bound of *current state* and the shape of *past states* comes from the fact that the agent can traverse twice to the same state before the episode is terminated, which was needed to account for.

Otherwise, if the environment used default observation limits, these were provided:

- Current state:
  - High bound = Maximum value a 64-bit integer can hold,
  - Shape = 1-dimensional
- Past states:
  - High bound = Maximum value a 64-bit integer can hold,
  - Shape = 100
- Transitions to states
  - High bound = Maximum value a 64-bit integer can hold,
  - Shape = 100
- Amount of states visited
  - High bound = Maximum value a 64-bit integer can hold
  - Shape = 1-dimensional

A decision was made to limit the amount of observed history of past states to 100, as well as the maximum amount of transitions. However, in case the agent moves past 100 states, the environment removes the state that is at the beginning of the past states history array, and stores the newest visited state at the end of the array.

4.6 Verification of the environment

A total of five exceptions has been implemented, three of which perform a pre-eliminary check of the provided CSV file (white space added for reading clarity):

- FSM Start State Not Found Error: This exception is thrown if there are no start states specified in the CSV file.
- FSM Transition State Not Found Error: This exception is thrown if the environment framework cannot find the state that the agent chose to traverse to.
- FSM Incorrect Preset Start State Error: This exception is thrown if a given preset start state is not set to be a start state, or if it does not exist in the CSV file.

- CSV File Transitions Error: This exception is thrown if the amount of transitions specified in any of the following three fields in any row of the CSV file does not match each other:
  - Possible discrete actions (transitions)
  - Transition names
  - Transitions to states
- CSV File Empty Values Error: This exception is thrown if there is any empty value in any field of any row.

4.7 Helper functions

Three helper functions have been implemented to help with the verification and initialization of the environment (white space added for reading clarity):

- Convert State Names(file_name:*str*): Adds a column of discretized state names to the CSV file provided if it does not contain such already, using the *Pandas* data analysis library [60]. If a state name repeats, it uses the same discretized state name, as the function assumes the states are the same in their functioning.
- Is Csv Empty(file_name:*str*): Checks if there is any empty value in any field of any row of the CSV file.
- Has States With Not Matching Transitions(file_name:*str*): Checks if the amount of transitions match in each field, as previously listed in the exceptions list.

Each of these functions is run during the initialization of an environment to perform preeliminary testing on the CSV file.

4.8 Deep reinforcement learning framework

To perform evaluation on the further implemented FSM environments, we decided to use the *Stable Baselines3* package, which is a set of well-tested implementations of deep reinforcement learning algorithms. In particular, we decided to use the PPO and A2C implementations, as they support the Dict gym observation spaces, and multiprocessing.

4.9 FSM environments

A decision was made to implement three finite state machine environments for the further evaluation of our framework and the chosen algorithms.

4.9.1 Pelican crossing

We implemented the pelican crossing FSM from Lloyd-Roberts's research [22] first, as although it is rather simple in its state space, and has an action space of two discrete actions, it has multiple start states, which we decided was interesting to evaluate whether the chosen algorithms could learn how to traverse through the state space while being initialized in random start states.

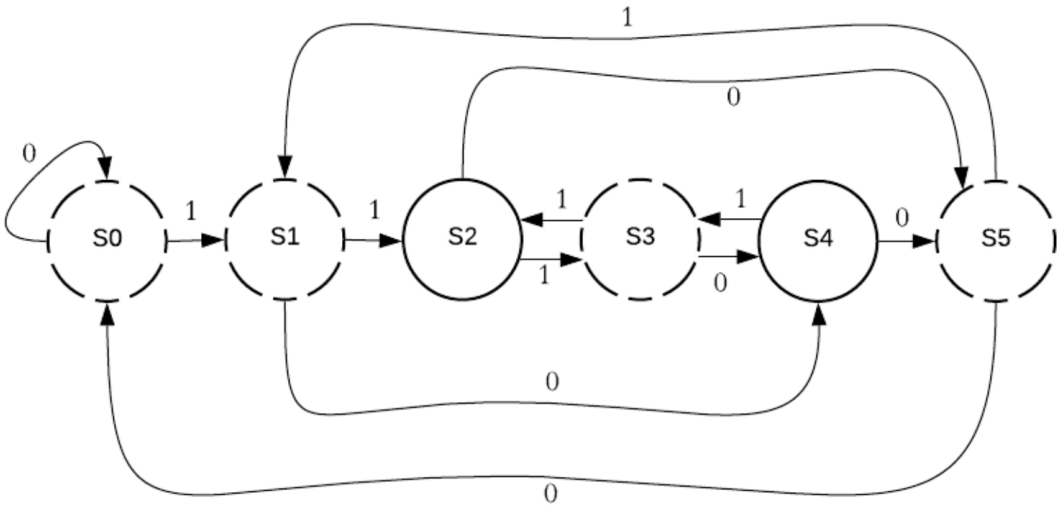


Figure 4.1: A finite state machine of a pelican crossing [22, p. 19].

We can define the finite state machine of a pelican crossing as follows:

$$
\begin{aligned}
&Q = \{S0, S1, S2, S3, S4, S5\},\\
&\Sigma = \{0, 1\},\\
&q_0 = \{S0, S1, S3, S5\},\\
&F = \emptyset.
\end{aligned}
$$

And the transition function δ:

Table 4.1: A state-transition table of a pelican crossing FSM.

Input	Current state	Next state
0	S0	S1
1	S0	S1
0	S1	S4
1	S1	S2
0	S2	S5
1	S2	S3
0	S3	S4
1	S3	S2
0	S4	S5
1	S4	S3
0	S5	S0
1	S5	S1

4.9.2 Toaster device

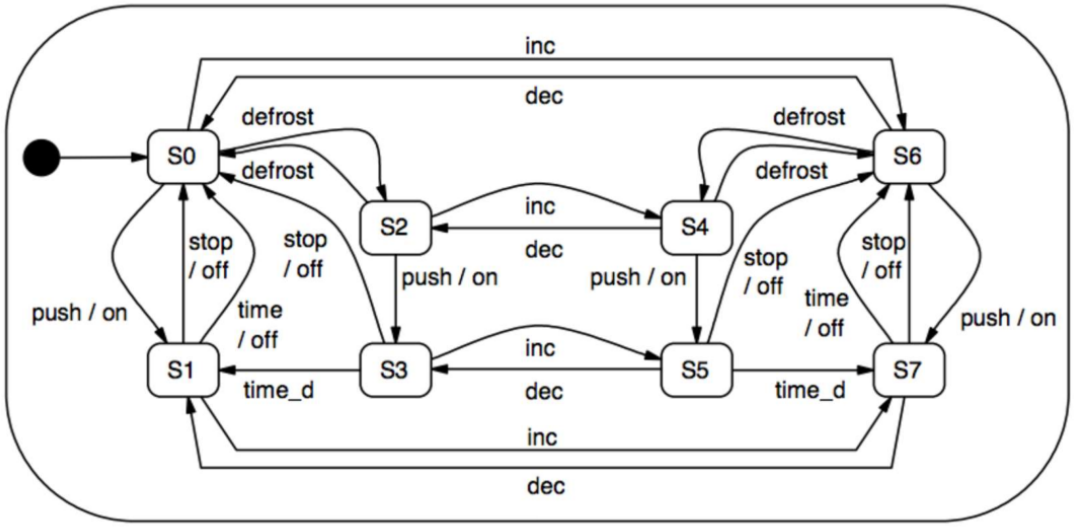


Figure 4.2: A finite state machine of a toaster [61, p. 81].

The next environment we decided to implement was a FSM of a toaster device [61, p. 81], as it has a larger state space, a larger action space of three discrete actions, and many close transitions between states that may trap the agent. It is important to note here that as we do not use the output of a transition in our environment, we omitted it from using it in our processed CSV file. We can define the finite state machine of a toaster device as follows:

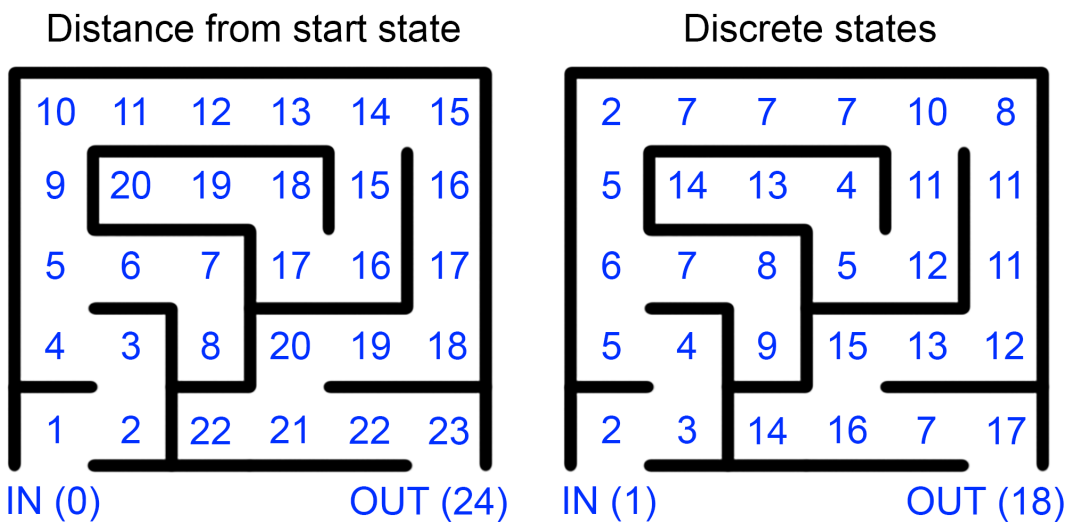
$$
\begin{aligned}
&Q = \{S0, S1, S2, S3, S4, S5, S6, S7\},\\
&\Sigma = \{dec, defrost, inc, push, stop, time, time_d\},\\
&q_0 = \{S0\},\\
&F = \emptyset.
\end{aligned}
$$

And the transition function δ:

Table 4.2: A state-transition table of a toaster device FSM.

Input	Current state	Next state
defrost	S0	S2
inc	S0	S6
push	S0	S1
inc	S1	S7
stop	S1	S0
time	S1	S0
defrost	S2	S0
inc	S2	S4
push	S2	S3
inc	S3	S5
stop	S3	S0
time_d	S3	S1
dec	S4	S2
defrost	S4	S6
push	S4	S5
dec	S5	S3
stop	S5	S6
time_d	S5	S7
dec	S6	S0
defrost	S6	S4
push	S6	S7
dec	S7	S1
stop	S7	S6
time	S7	S6

4.9.3 Maze



A. Distance of a state from the start state.

B. Discretized state names in the maze FSM.

Figure 4.3: Figures of the state space of the maze FSM.

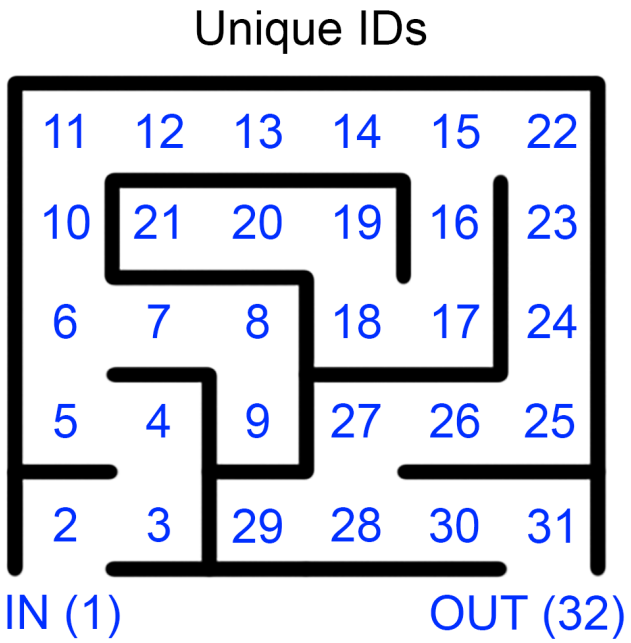


Figure 4.4: Unique IDs of states in the maze FSM.

Finally, we decided for our last FSM environment to be a maze. Here, we would be able to determine during evaluation whether the agent would be able to explore all of the states, especially when in states where there is more than one "valid" action, and eventually reach the end. To declare how the agent could move, we determined a simple limitation, which is that the agent is not able to perform a transition that would return it on its path. In states such as state (unique ID) 9, this means that all the agent can do when reaching the state 9 is perform a transition that would make it "hit the wall", which in our implementation means going to the same state, and therefore terminating the episode. We implemented a variable action space, as at states 1 and 32 (unique IDs), the agent can only perform one transition - in case of state 1, that is Up, which makes a transition to state 2, and in case of state 32, that is Left, which makes a transition to state 1, back to the start of the maze.

Moreover, as we have states that are similar, during the discretization of the state names we decided to duplicate some of the discrete state names. The best visible examples are of the states 6, 11, 12, 13, and 22. The similarities are that the agent transitions to each of the states from their left side and they also have only one valid action - to move right. Therefore, each of them received a discrete state name of 7.

We can define the finite state machine of the maze as follows:

$$
\begin{aligned}
&Q = \{1,2,3,4,5,6,7,8,9,\\
&10,11,12,13,14,15,16,17,18,19,\\
&20,21,22,23,24,25,26,27,28,29,30,31,32\},\\
&\Sigma = \{Up, Left, Down, Right\},\\
&q_0 = \{1\},\\
&F = \emptyset.
\end{aligned}
$$

We include the transition function δ as a state-transition table as Appendix A not to clutter the document.

4.10 Environment rendering wrapper

To visualize the paths of the agents for environments that do not violate the memory constraint, since the *gym* framework allows for the creation of wrappers to environments, we created a wrapper that counts the amount of visits to each state using a Python dictionary, except when

reset and starting in a start state, and, after a specified amount of timesteps, generates a heat map graph of the finite state machine. The graph also contains marked start states and the original transition labels. For this task of data visualization, we used the *Pydot* [62] package, which is a Python interface to *Graphviz*'s [63] dot language. This allowed us to program the generation of the graphs during the operation of the wrapper and the environment as a whole.

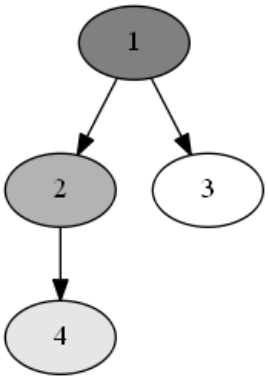


Figure 4.5: An example of a generated graph via the use of *Pydot*.

4.11 Implementation issues

During our implementation, we found out that the gym environment framework does not allow to change the action space during the training of agents. This meant that we had to change how our mechanic of the action space worked. Instead of changing the action space, we made it dependent on whether the user decided to scan the file for observation limits. If yes, the action space was set to the maximum amount of transitions any state had. If not, we set a hard limit of 10 for the amount of transitions to limit the amount of invalid actions in cases where the action space was small. The possibility of invalid actions however meant that we had to provide our rendering wrapper with information about whether the last tried transition was due to an invalid action or valid action, so it would not render incorrect edges on the generated graph. Moreover, the addition of the possibility of taking invalid actions has increased the risk of an agent staying in the same state for a prolonged period of time. This, however, was in at least a small scale mitigated by the use of multiple agents.

Furthermore, we decided to add multiprocessing for faster training and the possibility of boosting the convergence rate. Stable Baselines3 offers two types of vectorized environments, which allow to use multiple environments at the same time: DummyVecEnv and SubprocEnv

[64]. While the default way to use multiple environments by the Stable Baselines3 functions is the use of DummyVecEnv, in our case it did not work as DummyVecEnv runs multiple agents on the same environment, which resulted in agents receiving wrong observations as they tried to use the same variables at the same time, which resulted in a terrible training process, as each of them asynchronously updated the variables during the stepping function. Therefore, we decided to use SubprocEnv, which runs one agent on one environment. However, as it uses multithreading, it means that we are able to run at the same time for example, 8 environments on a CPU that has 8 logical cores. Unfortunately, after a small amount of testing, we realized that SubprocEnv did not wrap our environments with the rendering wrapper. This meant that we have had to restructure our rendering wrapper for a vectorized environment, which added further overhead to our project. The overhead also came from another issue related to using a vectorized environment, as when a vectorized environment terminates an episode and resets, the last observation given by an agent is of the already reset state. We eventually managed to overcome over all of these issues.

4.11.1 Multiprocessing and rendering

In the end, after resolving previously mentioned issues, our framework allows for multiprocessing by vectorizing an environment using the provided SubprocEnv by Stable Baselines3. Our rendering wrapper now counts the amount of visits, except when reset and starting in a start state, by each agent during training with multiprocessing.

4.12 Vectorized environment helper function

With the overhead of having to create a vectorized environment to properly use multiprocessing and/or rendering, we added a helper function to make it easier for the user to use our framework. The following function, Create Vec Fsm Env (white space added for reading clarity), returns a vectorized environment that bases on arguments passed to it:

- file name: The file name of the CSV file.
- num envs: How many environments should run at the same time.
- start state id: Allows for a preset start state ID. If 0, the start states will be randomized at each reset of the environment.

- use default obs space limits: If false, the environment will scan the file for limits. If true, it will use hard-coded limits.
- max row count: If the previous argument was False, this argument allows to use the default observation limits after hitting a set amount of scanned rows.
- wrapped: Allows to set whether to render the training process.

4.13 Hyperparameter tuning

After we ran our evaluation on default hyperparameters, the description of which can be found in Chapter 5, we used Optuna [24] to tune the hyperparameters for each environment, algorithm, and observation limit setting. We chose a number of hyperparameters for each algorithm and ran 50 trials of 10,000 time steps for each environment, algorithm, and observation limit setting, with a deterministic evaluation to know which hyperparameters performed best in 10,000 time steps. We ran 12 trials of 10,000 time steps and 50 runs, which brings our total of training time steps to 6,000,000 time steps.

4.13.1 Advantage Actor Critic

In the case of A2C, we chose the following hyperparameters to tune (white space added for reading clarity) [65]:

- gamma: The discount factor.
- max grad norm: Maximum value for the gradient clipping.
- gae lambda: Factor concerning the trade-off of bias vs variance.
- learning rate: The learning rate.
- ent coef: The entropy coefficient.
- vf coef: Value function coefficient (similar to the entropy coefficient).
- net arch: Whether to use networks consisting of two layers of 64 neurons for both the entropy and value function, or 256 neurons.
- activation fn: Whether to use activation function ReLU or tanh for the neurons.

4.13.2 Proximal Policy Optimization

In the case of PPO, we chose the following hyperparameters to tune (white space added for reading clarity) [66]:

- gamma: The discount factor.
- learning rate: The learning rate.
- ent coef: The entropy coefficient.
- n epochs: The amount of epochs to run.
- gae lambda: Factor concerning the trade-off of bias vs variance.
- max grad norm: Maximum value for the gradient clipping.
- vf coef: Value function coefficient (similar to the entropy coefficient).
- net arch: Whether to use networks consisting of two layers of 64 neurons for both the entropy and value function, or 256 neurons.
- activation fn: Whether to use activation function ReLU or tanh for the neurons.

Chapter 5

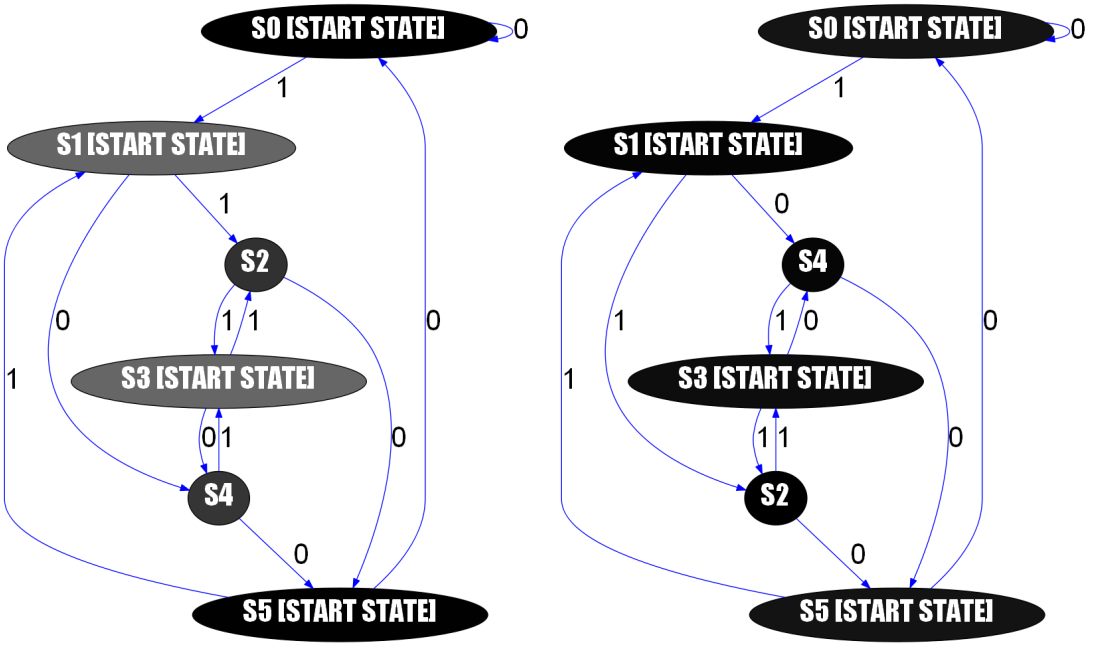
Evaluation

In this chapter, we evaluate the training progress on each of the environments. We ran each of the environments once with the rendering wrapper and 8 agents running at the same time to demonstrate the rendering wrapper and how do the chosen algorithms, A2C and PPO, behave on default hyperparameters. The amount of time steps chosen for this was 200,000. Afterwards, we ran 6 trials of 150,000 time steps and 20 runs of each algorithm and each environment, with the observation limits given by scanning the CSV file. We also ran 6 trials with the same amount of time steps and runs using the default, explicitly set observation limits. In total, we ran 12 trials of 150,000 time steps and 20 runs, which means that overall, we ran the training of models for 36,000,000 time steps.

5.1 Generated graphs

In this section, we will focus on the evaluation of graphs generated by the exploration of agents in each of the environments. Each training used observation limits provided from a file and used multiprocessing via 8 agents.

5.1.1 Pelican environment



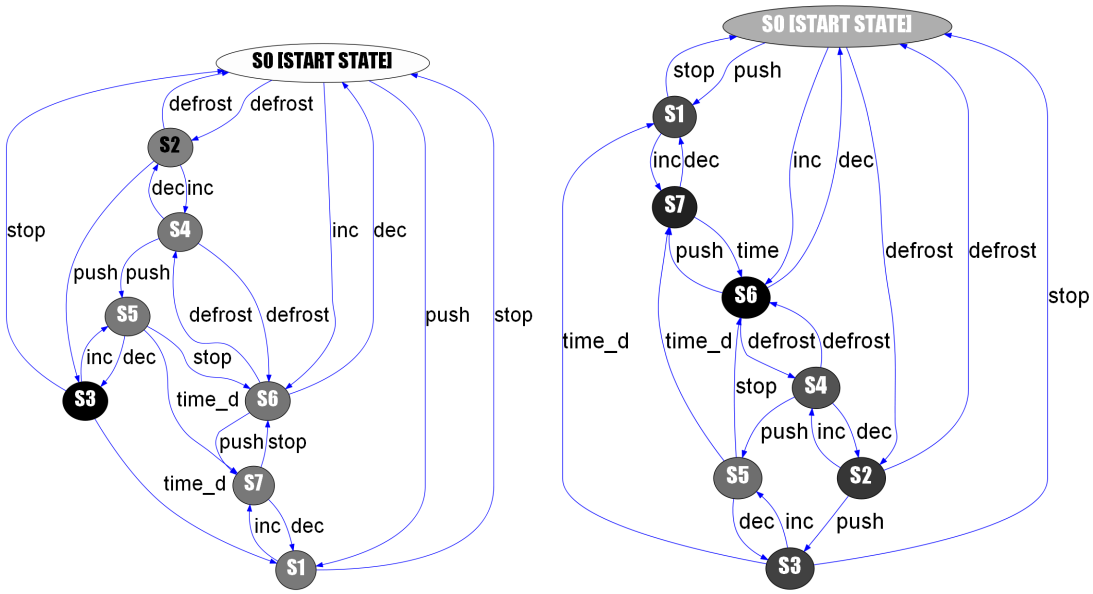
A. Graph generated by A2C.

B. Graph generated by PPO.

Figure 5.1: Graphs of the pelican environment generated by the exploration of A2C and PPO agents.

Both algorithms managed to successfully traverse through the entire state space, which shown solid ground for the further exploration of other environments. What is interesting to see at this stage is the difference between the colouring of the heat maps of A2C and PPO. As shown on Figure 5.1, A2C has generated the most movement in states $[S0, S2, S4, S5]$, which could either show a tendency to quickly converge to a set path, or the inability to handle the randomization of the start states. PPO on the other hand has pretty much generated the same amount of movement in each state, which could show an instability in convergence.

5.1.2 Toaster environment



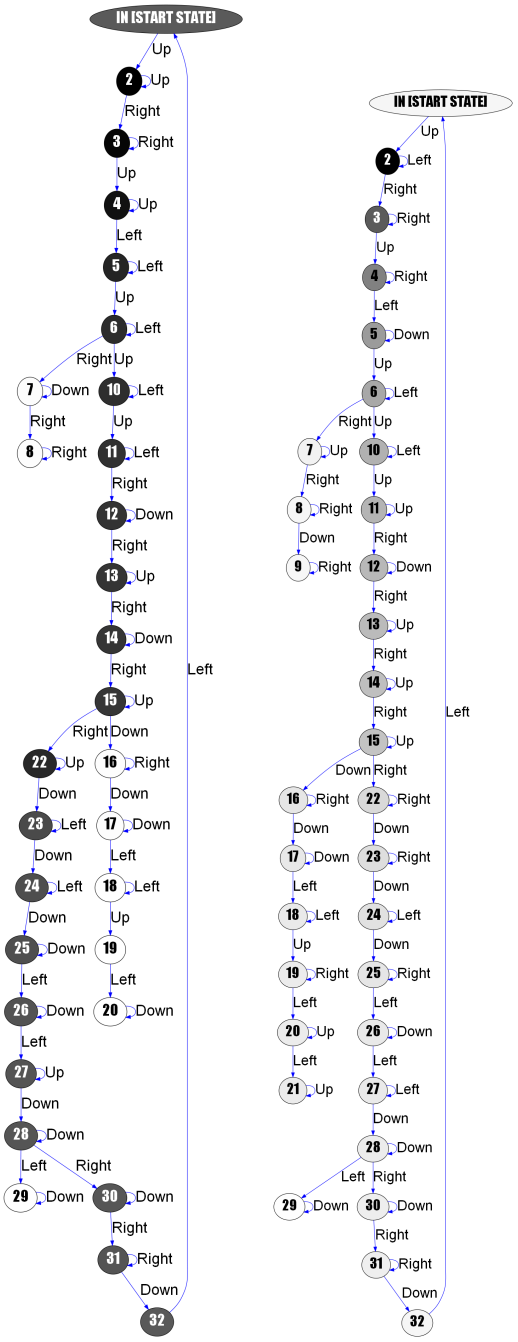
A. Graph generated by A2C.

B. Graph generated by PPO.

Figure 5.2: Graphs of the toaster environment generated by the exploration of A2C and PPO agents.

Again, both algorithms managed to successfully traverse through the entire state space. Here, we see an even clearer difference between A2C and PPO, which could be though due to a single, unfortunate, run with the rendering wrapper. As shown on Figure 5.2, A2C nearly does not ever step in the start state $S0$, and instead most likely finishes its paths somewhere in the entanglement of states. It is interesting to consider why did it decide not to consider traversing to the start state as a final move. PPO manages to perform far better in this example, as it has a more even amount of visits to each state.

5.1.3 Maze environment



A. Graph generated by A2C.

B. Graph generated by PPO.

Figure 5.3: Graphs of the maze environment generated by the exploration of A2C and PPO agents.

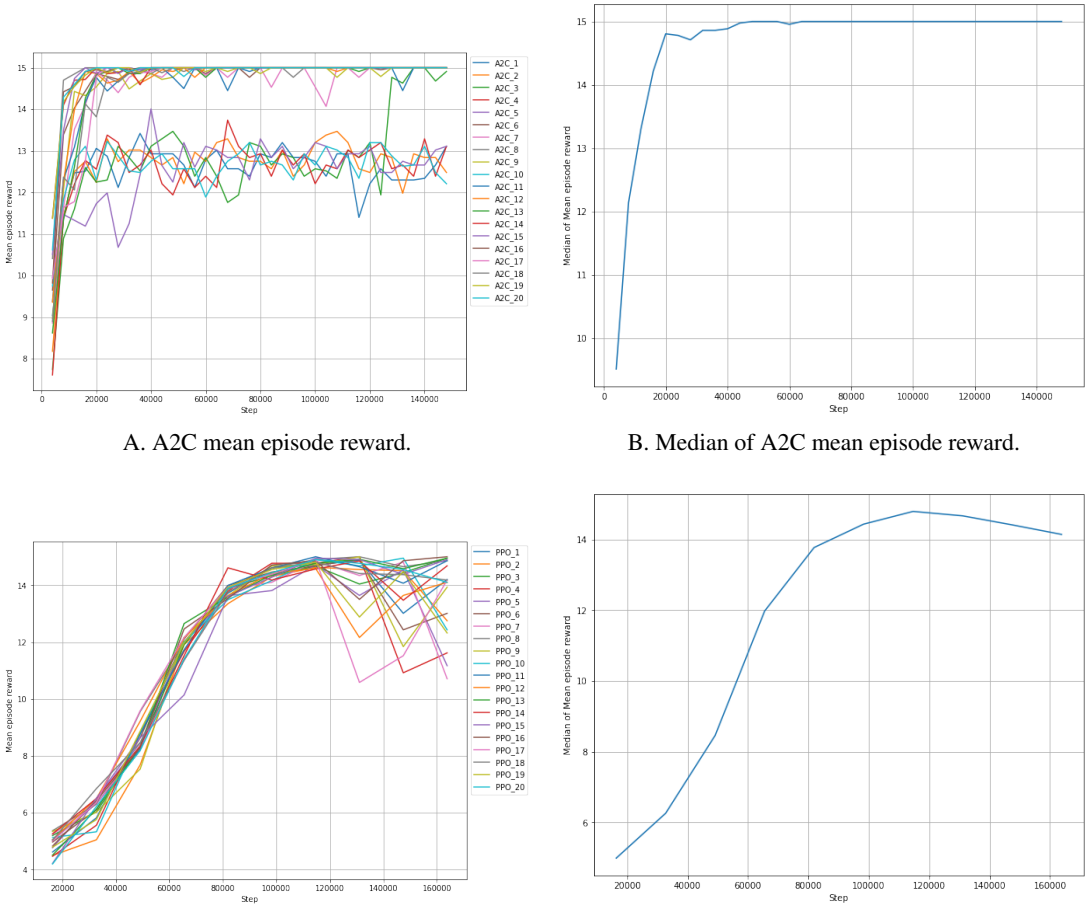
This is the first environment when an algorithm could not explore the entire state space. As shown on Figure 5.3 above, A2C did not manage to find the states $[9, 21]$, which PPO managed to find. On the other hand, A2C managed to find the optimal path through the states the quickest, as we can see it traversing much more often through the final state 32 than PPO, which we would like to indicate. Nonetheless, PPO performed better in regards of our goal of validating the state space, although it seems it took many more time steps to find the optimal path throughout the finite state machine than the A2C algorithm.

5.2 Performance comparison

In this section, we will demonstrate the results of the trials, and compare the performance of A2C and PPO in each environment. We will also see if the algorithms managed to successfully train the agents when using default observation limits.

5.2.1 Pelican environment

5.2.1.1 Using limits from scanning the file



A. A2C mean episode reward.

B. Median of A2C mean episode reward.

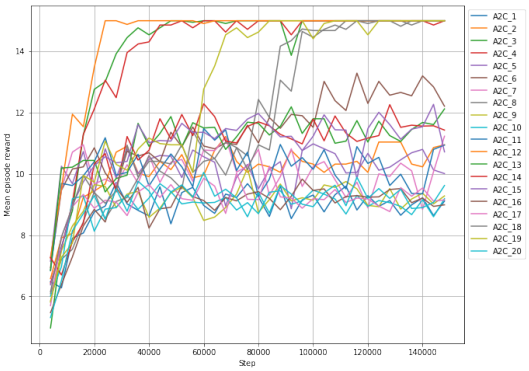
C. PPO mean episode reward.

D. Median of PPO mean episode reward.

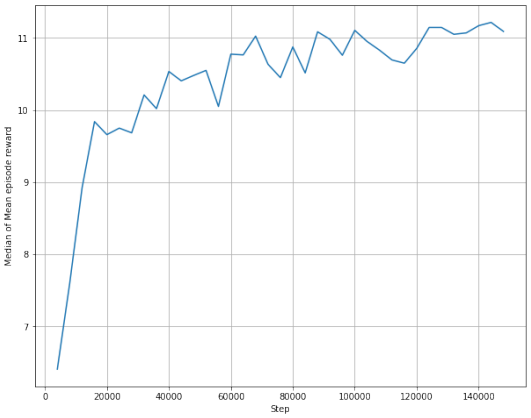
Figure 5.4: Plotted results of pelican trials that used observation limits based on the CSV file.

PPO here had a much more stable convergence than A2C, although it started to lose its stability in the end. While A2C managed to converge to an optimal policy much quicker than PPO, it had multiple runs that stayed at a local optimum. If we check the median plot however, it seems that A2C on average would be a better choice here.

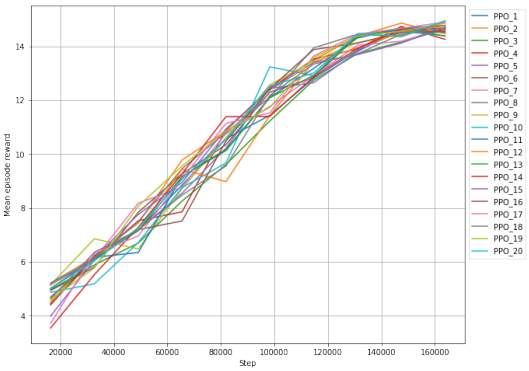
5.2.1.2 Default observation limits



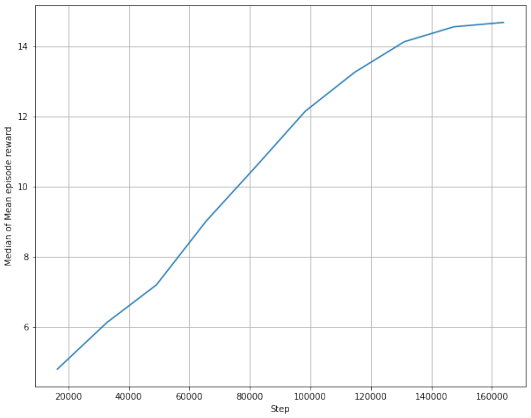
A. A2C mean episode reward.



B. Median of A2C mean episode reward.



C. PPO mean episode reward.



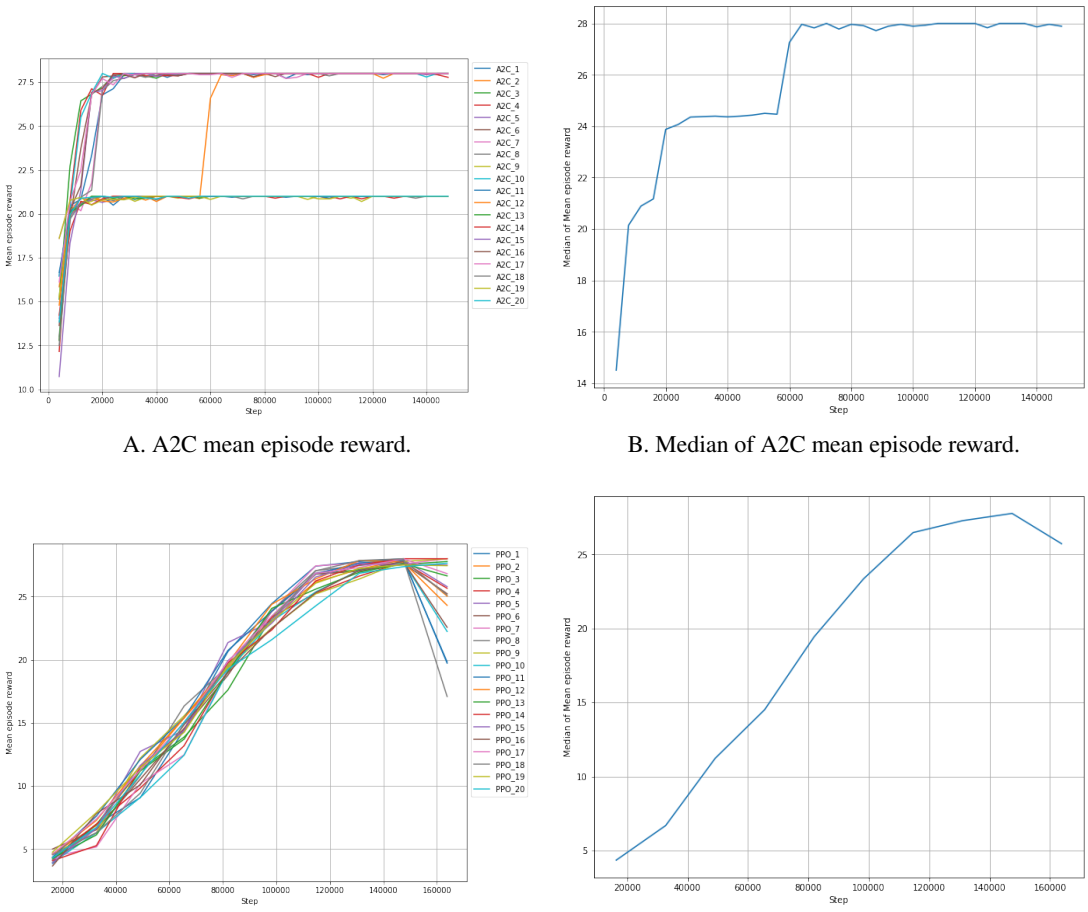
D. Median of PPO mean episode reward.

Figure 5.5: Plotted results of pelican trials that used default observation limits.

With the introduction of default observation limits, A2C started to be very unstable in its policy training as shown on the plots. PPO, on the other hand, managed to stay stable in its gradient descent. This shows the potential issues that can arise if we used A2C with an extremely large state space.

5.2.2 Toaster environment

5.2.2.1 Using limits from scanning the file



A. A2C mean episode reward.

B. Median of A2C mean episode reward.

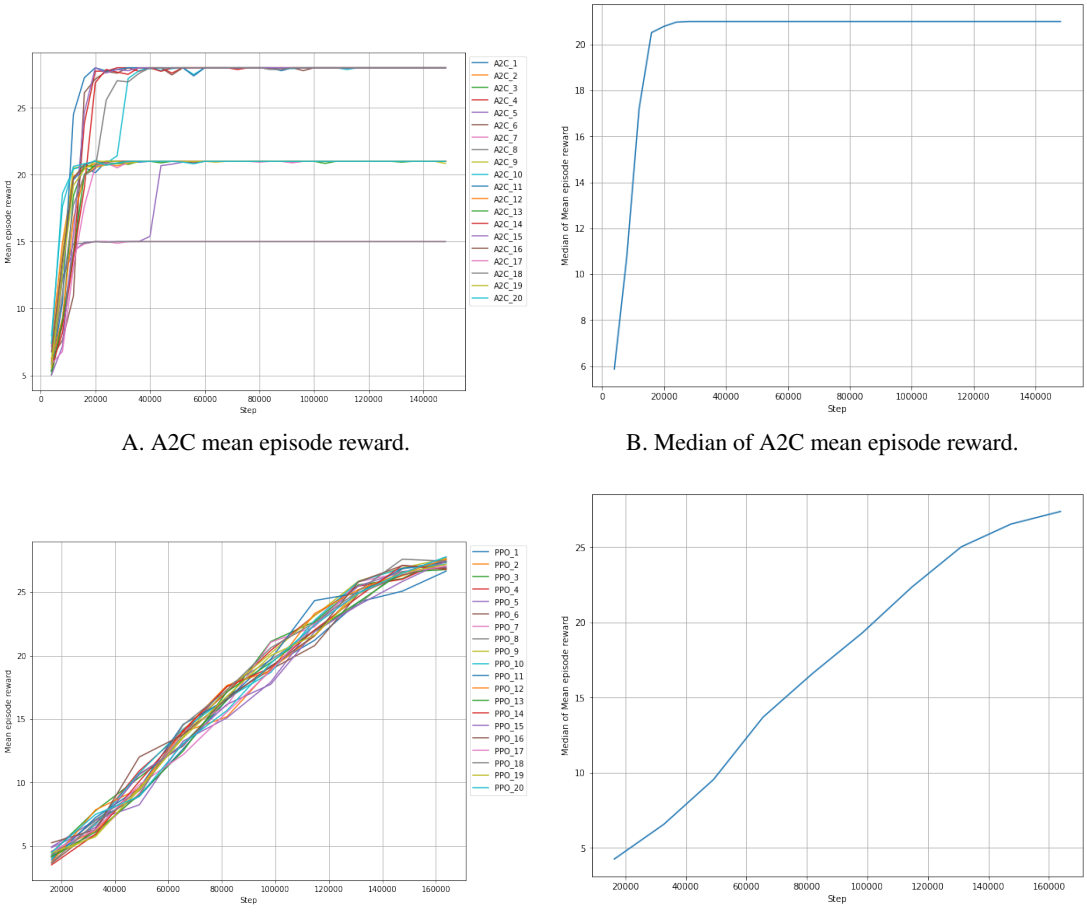
C. PPO mean episode reward.

D. Median of PPO mean episode reward.

Figure 5.6: Plotted results of toaster trials that used observation limits based on the CSV file.

Interestingly, the training runs on the toaster environment seem to be much more stable in case of both algorithms. The plot of A2C mean episode reward, Figure 5.6A., seems to explain the behaviour of the A2C agent in the rendered graph in Section 5.1.2, as many of the runs get stuck on a similar local optimum policy. Again, PPO is more stable in learning, but A2C, as a median, converges to the most optimal policy the quickest.

5.2.2.2 Default observation limits



A. A2C mean episode reward.

B. Median of A2C mean episode reward.

C. PPO mean episode reward.

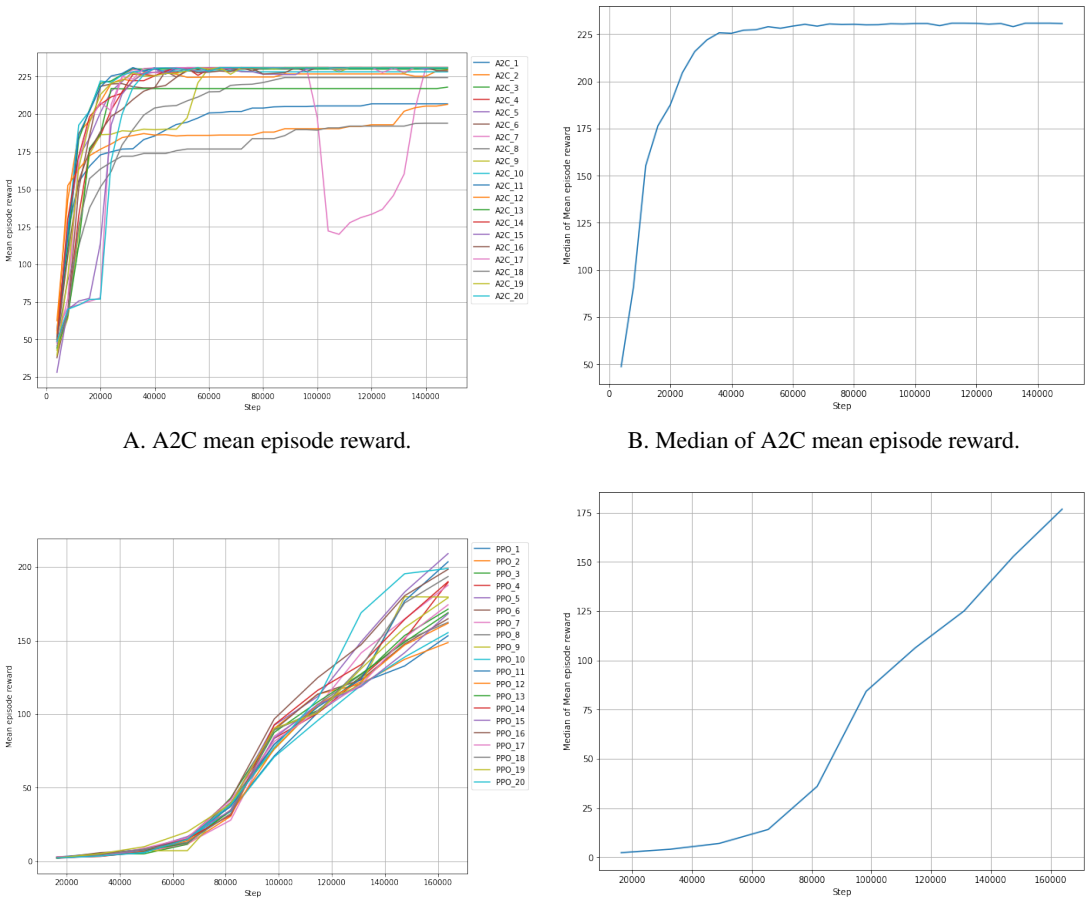
D. Median of PPO mean episode reward.

Figure 5.7: Plotted results of toaster trials that used default observation limits.

A2C manages to get stuck on a new local optimum, but strangely, in a behaviour different from the situation in the pelican environment evaluation on default observation limits, 5.2.1.2, remains stable and finds an optimal policy even faster than when using observation limits based on the CSV file. PPO remains stable in training, although it does encounter a small slowdown in the convergence.

5.2.3 Maze environment

5.2.3.1 Using limits from scanning the file



A. A2C mean episode reward.

B. Median of A2C mean episode reward.

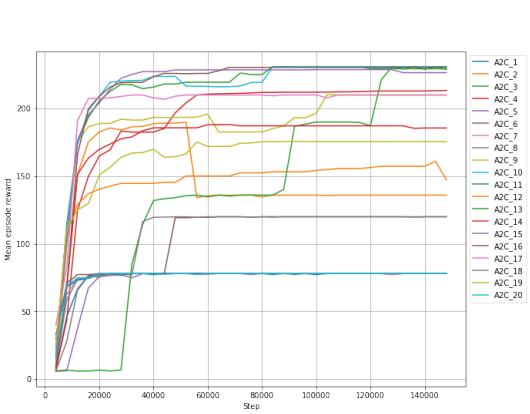
C. PPO mean episode reward.

D. Median of PPO mean episode reward.

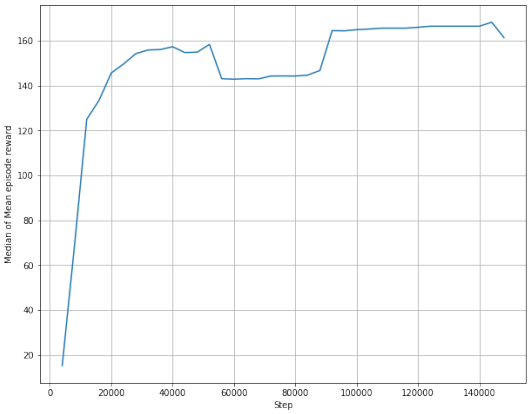
Figure 5.8: Plotted results of maze trials that used observation limits based on the CSV file.

The graphs from the Figure 5.8 above, similarly as in Section 5.2.2.1, seems to explain the performance of both algorithms when used with the rendering wrapper (Section 5.1.3). A2C finds the optimal path and furthermore, the optimal policy, much more quicker than PPO. That said, if PPO had about 50,000 time steps more, it would most likely converge to an approximately similar optimal policy.

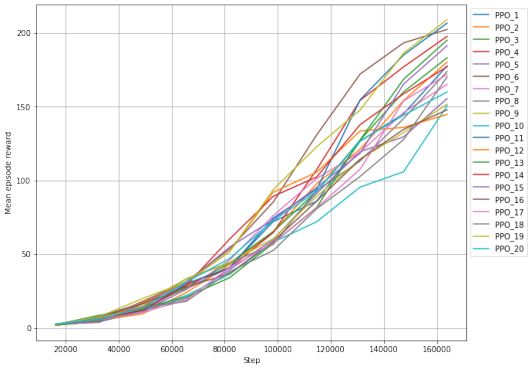
5.2.3.2 Default observation limits



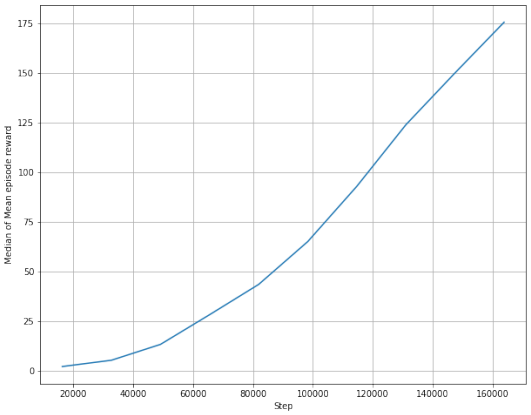
A. A2C mean episode reward.



B. Median of A2C mean episode reward.



C. PPO mean episode reward.



D. Median of PPO mean episode reward.

Figure 5.9: Plotted results of maze trials that used default observation limits.

The gathered performance of A2C yet again, as in Section 5.2.1.2, breaks down into multiple local optimal policies, while PPO retains stable. It should be said, however, that A2C still beats PPO time-wise in terms of finding the most optimal policy.

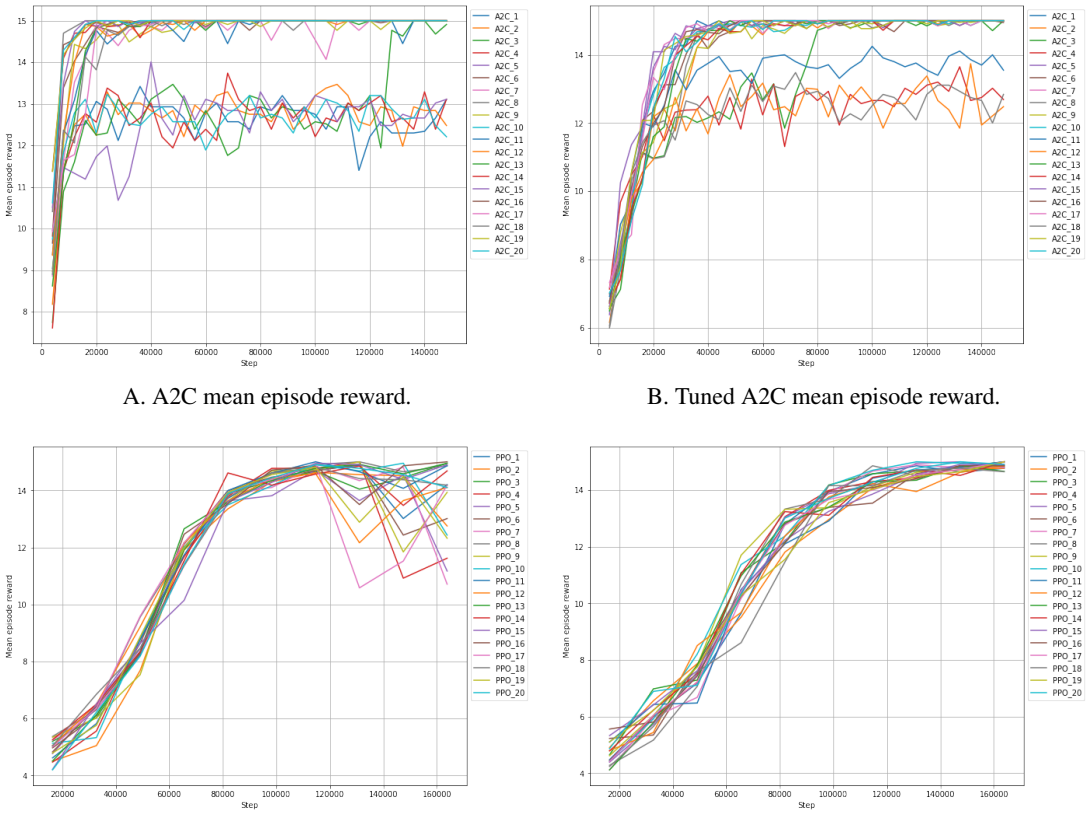
5.3 Tuned hyperparameters evaluation

To limit the amount of plots for evaluation, we will compare solely the plots of mean episode rewards, without the medians of them. This is to allow better clarity when reading this document and for better understanding of our evaluation. Similarly to the previous evaluations,

we ran 12 trials of 150,000 time steps and 20 runs using the algorithms with the auto-tuned hyperparameters via Optuna.

5.3.1 Pelican environment

5.3.1.1 Using limits from scanning the file



A. A2C mean episode reward.

B. Tuned A2C mean episode reward.

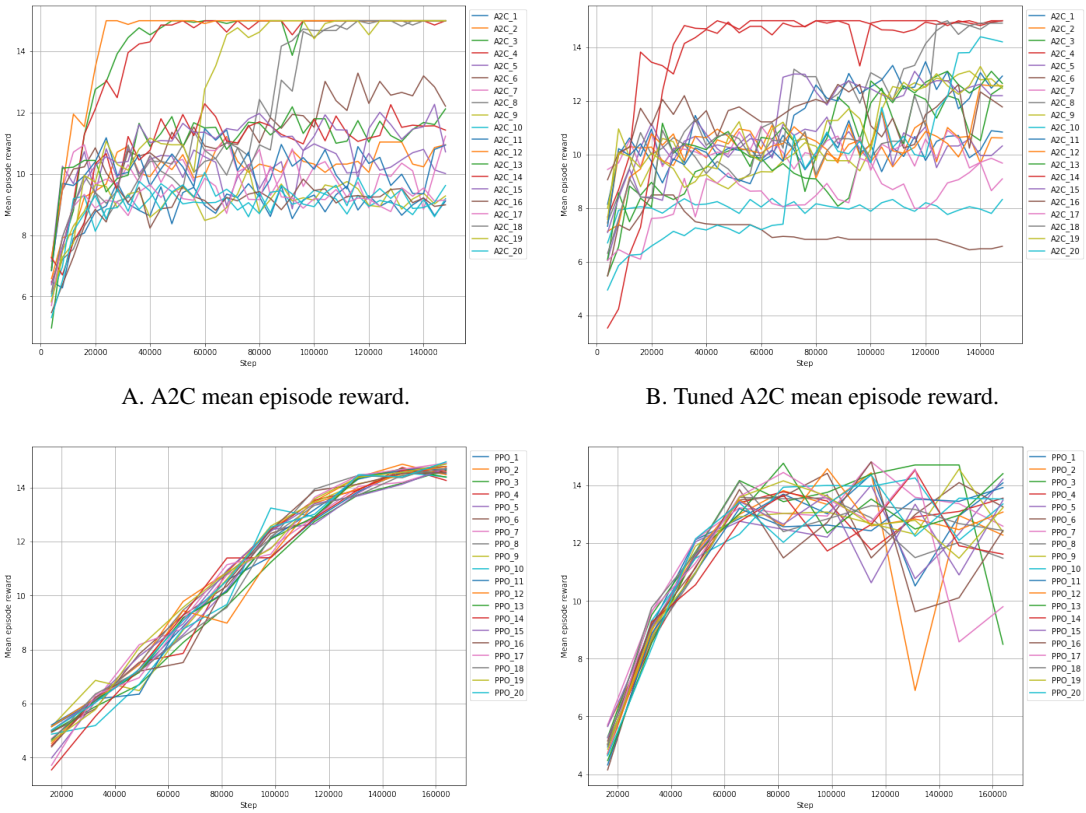
C. PPO mean episode reward.

D. Tuned PPO mean episode reward.

Figure 5.10: Plotted results of pelican trials, alongside results with tuned hyperparameters, that used observation limits based on the CSV file.

Curiously, the tuned hyperparameters resulted in a more stable training process for the PPO algorithm, while it actually worsened for A2C, making it slower to reach an optimal policy.

5.3.1.2 Default observation limits



A. A2C mean episode reward.

B. Tuned A2C mean episode reward.

C. PPO mean episode reward.

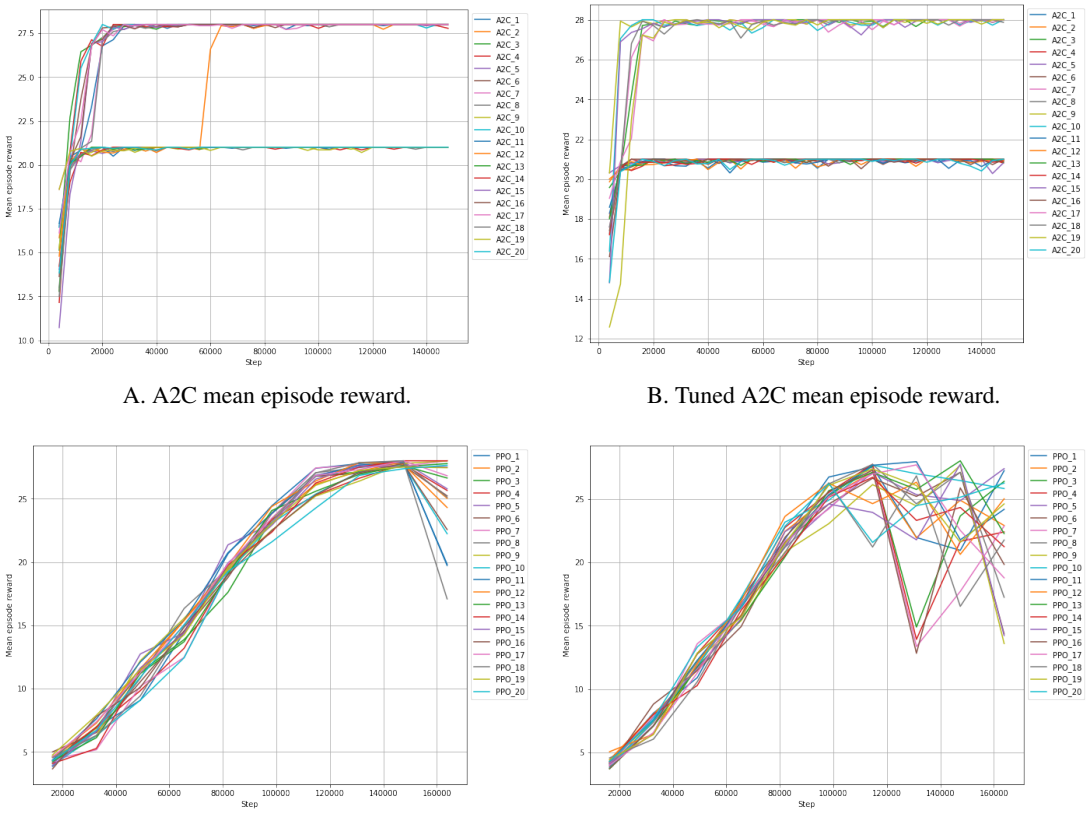
D. Tuned PPO mean episode reward.

Figure 5.11: Plotted results of pelican trials, alongside results with tuned hyperparameters, that used default observation limits.

Similarly to the previous Section 5.3.1.1, while in the case of A2C the tuning of the hyperparameters seemed to not make a difference, it made PPO converge much faster in the pelican environment setting, although each of the runs later strayed away from the optimal policy.

5.3.2 Toaster environment

5.3.2.1 Using limits from scanning the file



A. A2C mean episode reward.

B. Tuned A2C mean episode reward.

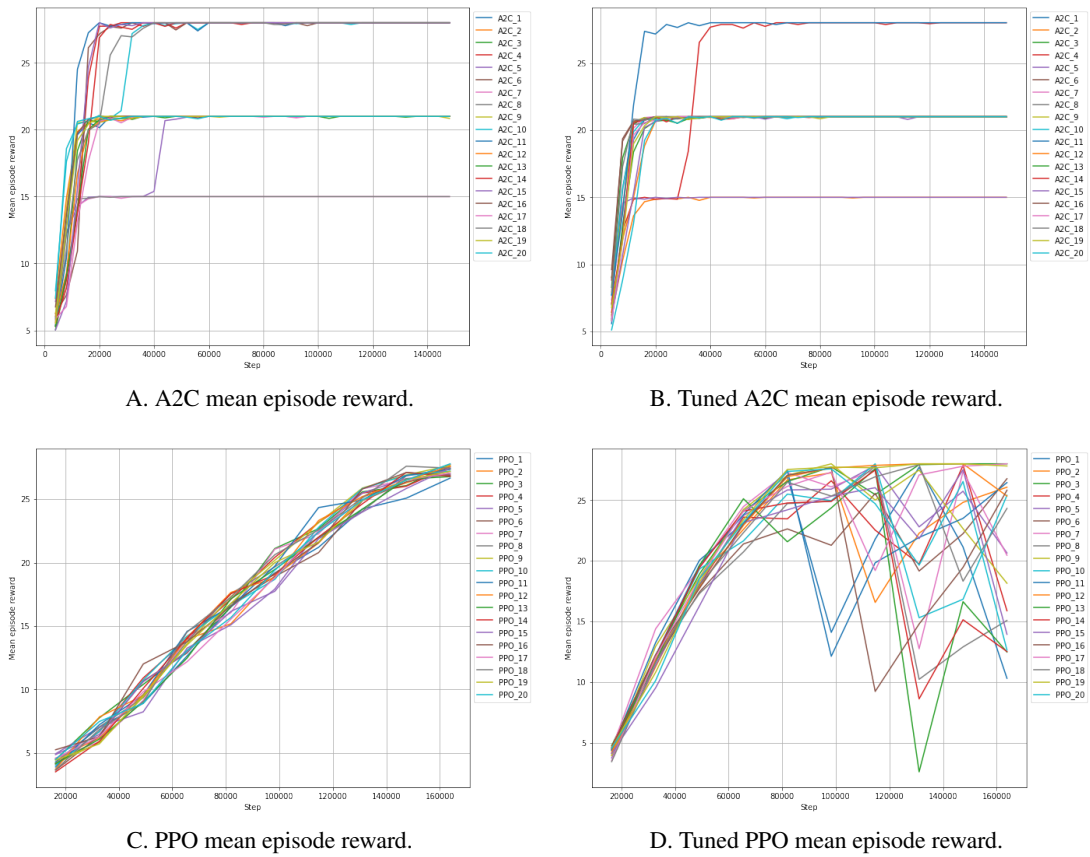
C. PPO mean episode reward.

D. Tuned PPO mean episode reward.

Figure 5.12: Plotted results of toaster trials, alongside results with tuned hyperparameters, that used observation limits based on the CSV file.

We can observe a small improvement in the convergence speed in both algorithms when using tuned hyperparameters, however, A2C has many more runs that got stuck in a local optimum than when using default hyperparameters.

5.3.2.2 Default observation limits



A. A2C mean episode reward.

B. Tuned A2C mean episode reward.

C. PPO mean episode reward.

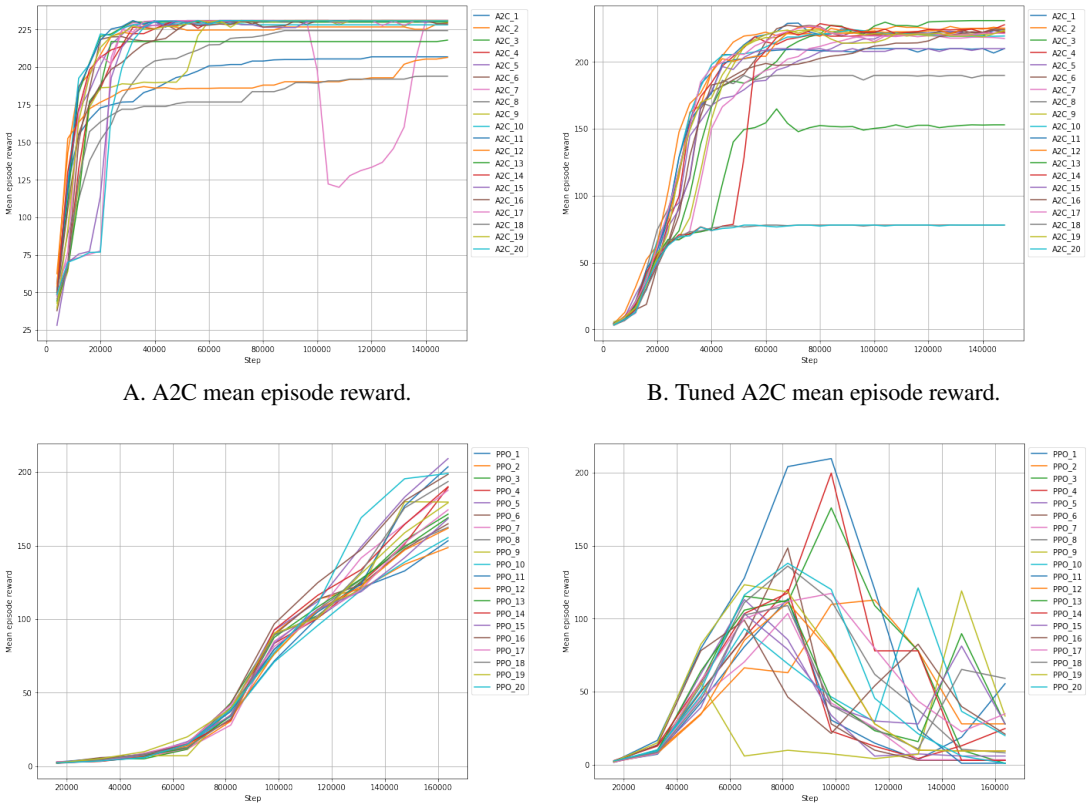
D. Tuned PPO mean episode reward.

Figure 5.13: Plotted results of toaster trials, alongside results with tuned hyperparameters, that used default observation limits.

The pattern from previous sections seems to be repeating. Tuned A2C again has more runs stuck in a local optimum compared to its performance with default hyperparameters, while PPO converges much faster to the optimal policy, although losing stability extremely quickly after doing so.

5.3.3 Maze environment

5.3.3.1 Using limits from scanning the file



A. A2C mean episode reward.

B. Tuned A2C mean episode reward.

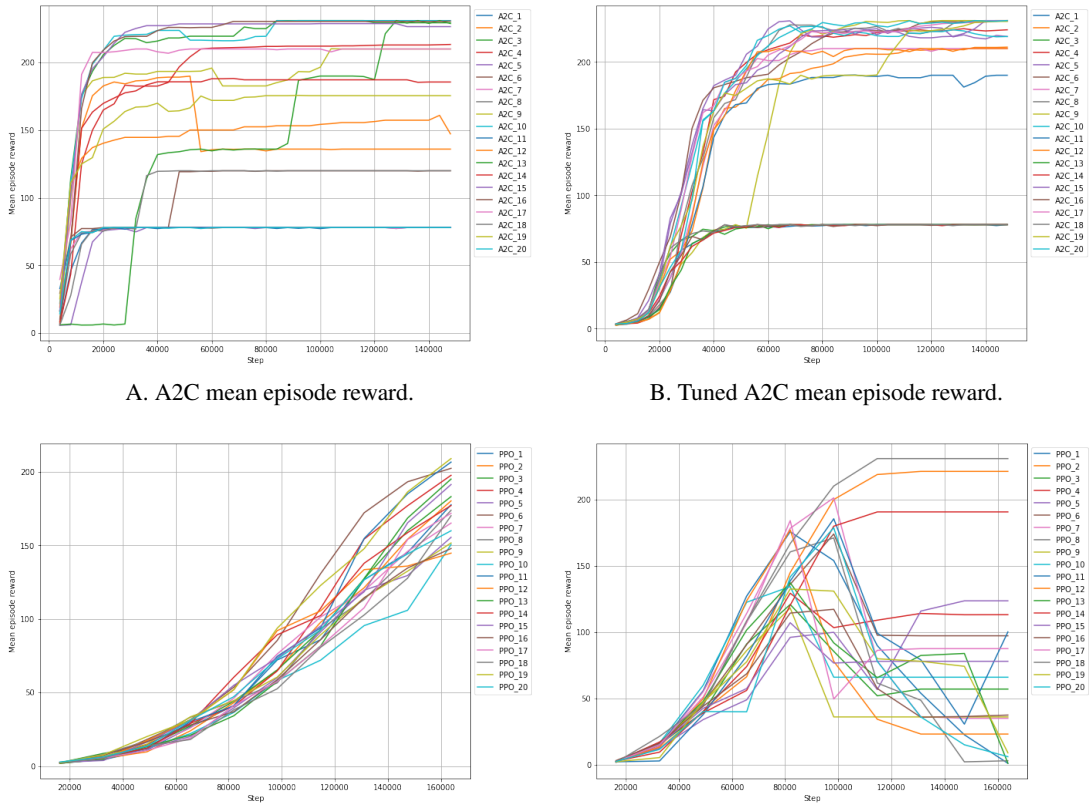
C. PPO mean episode reward.

D. Tuned PPO mean episode reward.

Figure 5.14: Plotted results of maze trials, alongside results with tuned hyperparameters, that used observation limits based on the CSV file.

While the performance of tuned A2C still remains worse to its default performance, we see a surprising discovery in the case of PPO. While we can observe that there have been some runs that received a massive convergence speed improvement, most of the runs got stuck on a local optimum and actually quickly worsened in performance. A2C comes on top in this scenario.

5.3.3.2 Default observation limits



A. A2C mean episode reward.

B. Tuned A2C mean episode reward.

C. PPO mean episode reward.

D. Tuned PPO mean episode reward.

Figure 5.15: Plotted results of maze trials, alongside results with tuned hyperparameters, that used default observation limits.

Yet again, just as in the previous Section 5.3.3.1, A2C remains a better choice in this environment setting while still having a worse performance than its default hyperparameters settings. We can repeatedly see a few runs that actually gain a massive improvement in the case of PPO, but the majority actually suffers from the tuned hyperparameters.

5.4 Summary

As we could see in the previous sections that showed comparisons of the performances of algorithms using plots of mean episode reward, the performance of a deep reinforcement learning algorithm can be easily swayed by any change of their "surroundings", whether that be a change both of and in the environment, or of their hyperparameters. Since we ran multiple runs in each

trial, we were able to demonstrate that the performance of deep reinforcement learning algorithms varies highly, especially when modifying the hyperparameters they use for their internal training algorithm. The performance of algorithms using tuned hyperparameters could also be due to the fact that although we ran 50 trials for each algorithm in each case, we only used 10,000 time steps. This could be a potential reason to the fluctuations of the results we gathered, and furthermore be a reason why PPO only sometimes managed to gain a massive boost in its convergence speed.

Furthermore, it seems that when using default hyperparameters, in most cases A2C manages to find an optimal policy much quicker than PPO, and is a better solution when having the aim of maximizing the cumulative expected reward in mind. However, with our goal of exploring the state spaces of finite automata, we would most likely suggest the usage of PPO to have the better possibility of traversing through each state in the state space.

The main lesson to gather from this evaluation is that when training deep RL algorithms for later use, it is important to always evaluate a model during its training and save the best one, as due to their instability that comes with the use of deep neural networks, we can easily lose the optimal policy convergence.

Chapter 6

Conclusion

Deep reinforcement learning algorithms are powerful, although a bit unstable, tools that, as we have shown by our work, can be used in many fields. In this document, we have described the area of formal system modelling and the challenges related to the validation of such. Furthermore, we have reported on deep reinforcement learning, and how it can be used in areas that base on interaction. We also pointed to the lack of an existing reinforcement learning framework for validating finite state machines. With the gathered knowledge, we managed to successfully establish a framework for exploring the state spaces of finite state machines. We implemented three general finite state machines: a pelican crossing, where we also based on the ground work made by Lloyd-Roberts [22], a toaster device, and a maze. Both of the deep reinforcement learning algorithms that were used, Advantage Actor Critic and Proximal Policy Optimization, managed to explore the state spaces of the provided finite state machines. With the further evaluation we have done on the implemented finite state machines, we determined that the default implementation of PPO, provided by Stable Baselines3, performs better in the task of exploring the state spaces, albeit A2C could potentially be more adequate if we planned on finding the longest path throughout the state space with the least amount of time steps, without the need for exploration. We also found that using algorithms with auto-tuned hyperparameters does not necessarily mean that we will gain an improvement in performance. The created rendering wrapper for our custom gym environment managed to generate a sufficient heat map visualization of the main states the agents were traversing through. While we have expected the incoming implementation obstacles from combining file operation, data visualization and reinforcement learning together, the main issues that we had to deal during the development of this project came unexpected, although we managed to successfully come

through them. Nonetheless, we are satisfied with achieving our goal of providing an operable RL framework for deterministic finite automata.

However, there are a few small disadvantages of our project that we wish to reflect upon here. While we have provided environments that were complex enough to suffice the evaluation of our framework, such as the maze environment, we were not able provide a finite state machine of an extremely large state space, for example, a FSM that converted to a CSV file would weigh 20 gigabytes. Although generating one was considered, we decided to append it as possible future work for this framework, as validating a finite state machine that we are simply unable to have a clear view of beforehand seemed improper. Instead, we decided to account for such future work by providing the default observation limits.

Moreover, during the part of auto-tuning hyperparameters of algorithms in this project, a small issue with invalid actions was found. As all runs of the algorithms during their tuning were run using a single agent, they were not supplemented by the multiprocessing exploration benefit. Therefore, many tuning trial runs, that used default observation limits or were run in environments containing invalid actions, could not move from the start state, and therefore were stuck for the rest of the episode. As this behaviour was not noted to be an issue in normal training, we have wrapped the tuning with a TimeLimit wrapper to prune trial runs that did not finish an episode by a set limit. Still, we would suggest for future work on this framework to consider terminating episodes where an invalid action was used multiple times in a row, instead of letting them run. The framework also does not consider a case where an user deliberately sets an amount of possible transitions to 0, which would most likely result in an episode running forever.

Another issue was found with the rendering package we used, *Pydot*, during the evaluation of the training on the maze environment. We found that during the visualization of the paths taken by agents, the visualization package does not draw multiple edges (with different transition labels) to the same node. The solution for this could be the allowance for multiple, similar edges, although this would require to keep the drawn edges stored in memory for ensuring that an edge does not get drawn multiple times, which would be a concerning disadvantage.

Concerning future work in particular, although this project had a large amount of evaluation done in particular, there is an almost tangible lack of testing value-based algorithms, such as Deep Q-Learning. Albeit in our case we were limited by the fact that only a small amount of implemented algorithms support the Dict observation space, perhaps we could find that other algorithms perform better in the task of exploration. Furthermore, while the project was limited

by the memory constraint and therefore decided not to pursue this path in case of potential further implications, a model-based approach most likely could be used in our framework, since we are able to predict in which state the agent will be after performing action A_t when in state S_t. This would in all probability require a complete implementation from scratch of a RL algorithm, however we assume that the sample-efficiency would highly improve in such case. Additionally, we also observe that there is a potential in expanding this framework to other, more expanded types of finite automata, such as extended finite state machines. Even the implementation of Moore and Mealy machines would be already a benefit to this project.

To conclude this chapter and this research as a whole, as we have finished the successful implementation of this research, we hope to set a stepping stone in the validation of models in hardware and software, as well as show the potential of exploration of state spaces via the use of PPO and A2C algorithms. The potentially biggest benefit we would like to end with is the fruitful usage of deep reinforcement learning, as it is an ever growing field of artificial intelligence, with potentially many more areas of usage to come.

Bibliography

[1] T. Broekel, "Measuring technological complexity - current approaches and a new measure of structural complexity," 2018.

[2] P. Deane, *The First Industrial Revolution*, 2nd ed. Cambridge University Press, 2000.

[3] G. Oregan, *Introduction to the History of Computing: A Computing History Primer*. Springer International Publish, 2016.

[4] J. Ryan, *A History of the Internet and the Digital Future*. Reaktion Books, 2010.

[5] S. Madakam, R. Ramaswamy, and S. Tripathi, "Internet of things (iot): A literature review," *Journal of Computer and Communications*, vol. 03, no. 05, p. 164–173, Jan 2015. [Online]. Available: https://www.scirp.org/html/56616_56616.htm

[6] W. Chaiyasoonthorn, K. Najantong, and S. Chaveesuk, "User perspective on the generation gap in using internet of things – iots: A conceptual framework," *Proceedings of 2019 the 9th International Workshop on Computer Science and Engineering*, p. 891–896, Jun 2019. [Online]. Available: http://www.wcse.org/content-12-136-1.html

[7] P. Britt, "Report: Connected devices have more than doubled since 2019," Jun 2021. [Online]. Available: https://www.telecompetitor.com/report-connected-devices-have-more-than-doubled-since-2019/

[8] J. McCarthy, "What is artificial intelligence?" Nov 2007. [Online]. Available: http://jmc.stanford.edu/articles/whatisai.html

[9] BuiltIn, "Artificial intelligence. what is artificial intelligence? how does ai work?" [Online]. Available: https://builtin.com/artificial-intelligence

[10] A. Dey, "Machine learning algorithms: A review," *International Journal of Computer Science and Information Technologies*, vol. 7, no. 3, p. 1174–1179, 2016. [Online]. Available: https://ijcsit.com/docs/Volume7/vol7issue3/ijcsit2016070332.pdf

[11] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. F. R. Jesus, R. F. Berriel, T. M. Paixão, F. W. Mutz, T. Oliveira-Santos, and A. F. de Souza, "Self-driving cars: A survey," *CoRR*, vol. abs/1901.04407, 2019. [Online]. Available: http://arxiv.org/abs/1901.04407

[12] U. J. Reddy, B. R. V. R. Reddy, and B. E. Reddy, "Recognition of lung cancer using machine learning mechanisms with fuzzy neural networks," *Traitement du Signal*, vol. 36, no. 1, p. 87–91, Apr 2019. [Online]. Available: https://www.iieta.org/journals/ts/paper/10.18280/ts.360111

[13] D. Yu and L. Deng, *Automatic Speech Recognition: A Deep Learning Approach*, 1st ed. Springer, 2015.

[14] J. Chen and W. K. Jenkins, "Facial recognition with pca and machine learning methods," *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Oct 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8053088

[15] X. Ning, F. Nan, S. Xu, L. Yu, and L. Zhang, "Multi-view frontal face image generation: A survey," *Concurrency and Computation: Practice and Experience*, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6147

[16] X. Tan, T. Qin, F. Soong, and T.-Y. Liu, "A survey on neural speech synthesis," 2021. [Online]. Available: https://arxiv.org/abs/2106.15561

[17] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *CoRR*, vol. abs/1811.12560, 2018. [Online]. Available: http://arxiv.org/abs/1811.12560

[18] Y. Li, "Deep reinforcement learning," *CoRR*, vol. abs/1810.06339, 2018. [Online]. Available: http://arxiv.org/abs/1810.06339

[19] I. A. Grout, *Digital Systems Design with FPGAs and CPLDs*. Elsevier Newnes, 2008.

[20] S. Kandar, *Introduction to Automata Theory, Formal Languages and Computation*. Dorling Kindersley (India), 2013.

[21] L. Harries, R. S. Clarke, T. Chapman, S. V. P. L. N. Nallamalli, L. Özgür, S. Jain, A. Leung, S. Lim, A. Dietrich, J. M. Hernández-Lobato, T. Ellis, C. Zhang, and K. Ciosek, "DRIFT: deep reinforcement learning for functional software testing," *CoRR*, vol. abs/2007.08220, 2020. [Online]. Available: https://arxiv.org/abs/2007.08220

[22] B. Lloyd-Roberts, "Applying reinforcement learning to railway interlocking verification," Master's thesis, Swansea University, Sept 2020.

[23] M. Hasanbeig, N. Y. Jeppu, A. Abate, T. Melham, and D. Kroening, "Deepsynth: Program synthesis for automatic task segmentation in deep reinforcement learning," *CoRR*, vol. abs/1911.10244, 2019. [Online]. Available: http://arxiv.org/abs/1911.10244

[24] Optuna, "A hyperparameter optimization framework." [Online]. Available: https://optuna.org/

[25] Merriam-Webster, "Automaton definition & meaning." [Online]. Available: https://www.merriam-webster.com/dictionary/automaton

[26] P. Linz, *An Introduction to Formal Languages and Automata*, 6th ed. Jones Bartlett Learning, 2017.

[27] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata theory, Languages, and Computation*, 3rd ed. Pearson/Addison Wesley, 2007.

[28] M. T. Morazán, J. M. Schappel, and S. Mahashabde, "Visual designing and debugging of deterministic finite-state machines in fsm," *Electronic Proceedings in Theoretical Computer Science*, vol. 321, p. 55–77, Aug 2020. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.321.4

[29] E. M. Clarke, T. A. Henzinger, and H. Veith, *Handbook of Model Checking*. Springer., 2018.

[30] P. M. Htut, K. Cho, and S. R. Bowman, "Grammar induction with neural language models: An unusual replication," *CoRR*, vol. abs/1808.10000, 2018. [Online]. Available: http://arxiv.org/abs/1808.10000

[31] V. Muralidaran, I. Spasić, and D. Knight, "A systematic review of unsupervised approaches to grammar induction," *Natural Language Engineering*, vol. 27, no. 6, p. 647–689, 2021. [Online]. Avail-

able: https://www.cambridge.org/core/journals/natural-language-engineering/article/abs/systematic-review-of-unsupervised-approaches-to-grammar-induction/

[32] D. A. Kolb, *Experiential learning: experience as the source of learning and development*. Pearson Education, 2015.

[33] K. C. Berridge and M. L. Kringelbach, "Pleasure systems in the brain," *Neuron*, vol. 86, no. 3, p. 646–664, 2015. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4425246/

[34] RainerGewalt, "Supervised vs unsupervised vs reinforcement learning - knowing the differences is a fundamental part of properly understanding machine learning," Jan 2021. [Online]. Available: https://starship-knowledge.com/supervised-vs-unsupervised-vs-reinforcement

[35] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2020.

[36] S. Yang, T. Ting, K. Man, and S.-U. Guan, "Investigation of neural networks for function approximation," *Procedia Computer Science*, vol. 17, pp. 586–594, 2013, first International Conference on Information Technology and Quantitative Management. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050913002093

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

[38] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning and Data Mining*, 2nd ed. Springer, 2017.

[39] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, p. 50–56, Nov 2016. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3005745.3005750

[40] OpenAI, "Part 2: Kinds of rl algorithms," 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[41] A. Brunnbauer, L. Berducci, A. Brandstätter, M. Lechner, R. M. Hasani, D. Rus, and R. Grosu, "Model-based versus model-free deep reinforcement learning for autonomous racing cars," *CoRR*, vol. abs/2103.04909, 2021. [Online]. Available: https://arxiv.org/abs/2103.04909

[42] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[43] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[44] L. Harries, S. Lee, J. Rzepecki, K. Hofmann, and S. Devlin, "Mazeexplorer: A customisable 3d benchmark for assessing generalisation in reinforcement learning," *2019 IEEE Conference on Games (CoG)*, p. 1–4, Sep 2019. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8848048

[45] H. Yang, X.-Y. Liu, S. Zhong, and A. Walid, "Deep reinforcement learning for automated stock trading: An ensemble strategy," *SSRN Electronic Journal*, Sep 2020. [Online]. Available: https://ssrn.com/abstract=3690996

[46] Y. Wu, E. Mansimov, S. Liao, A. Radford, and J. Schulman, "Openai baselines: Acktr a2c," Aug 2017. [Online]. Available: https://openai.com/blog/baselines-acktr-a2c/

[47] StableBaselines3, "Ppo," 2020. [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

[48] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015. [Online]. Available: http://arxiv.org/abs/1502.05477

[49] OpenAI, "Proximal policy optimization," 2018. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html

[50] Z. Ahmed, N. L. Roux, M. Norouzi, and D. Schuurmans, "Understanding the impact of entropy on policy optimization," *CoRR*, vol. abs/1811.11214, 2018. [Online]. Available: http://arxiv.org/abs/1811.11214

[51] G. Dulac-Arnold, R. Evans, P. Sunehag, and B. Coppin, "Reinforcement learning in large discrete action spaces," *CoRR*, vol. abs/1512.07679, 2015. [Online]. Available: http://arxiv.org/abs/1512.07679

[52] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Dec 2012. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6386109

[53] Python, "About python." [Online]. Available: https://www.python.org/about/

[54] Jupyter, "Jupyter notebook." [Online]. Available: https://jupyter.org/index.html

[55] Microsoft, "Visual studio code - code editing. redefined," Nov 2021. [Online]. Available: https://code.visualstudio.com/

[56] Anaconda, "The worlds most popular data science platform." [Online]. Available: https://www.anaconda.com/

[57] OpenAI, "A toolkit for developing and comparing reinforcement learning algorithms." [Online]. Available: https://gym.openai.com/

[58] ——, "gym/dict.py at master · openai/gym," Nov 2021. [Online]. Available: https://github.com/openai/gym/blob/master/gym/spaces/dict.py

[59] ——, "gym/box.py at master · openai/gym," Nov 2021. [Online]. Available: https://github.com/openai/gym/blob/master/gym/spaces/box.py

[60] pandas. [Online]. Available: https://pandas.pydata.org/

[61] J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-based testing for embedded systems*. CRC Press, 2012.

[62] E. Carrera. [Online]. Available: https://pypi.org/project/pydot/

[63] Graphviz. [Online]. Available: https://www.graphviz.org/

[64] StableBaselines3, "Vectorized environments." [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/guide/vec_envs.html

[65] ——, "A2c." [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html#parameters

[66] ——, "Ppo." [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html#parameters

Appendix A

State-transition diagram of maze FSM

Table A.1: A state-transition table of the maze FSM.

Input	Current state	Next state
Up	1	2
Up	2	2
Left	2	2
Right	2	3
Up	3	4
Down	3	3
Right	3	3
Up	4	4
Left	4	5
Right	4	4
Up	5	6
Left	5	5
Down	5	5
Up	5	6
Left	5	5
Down	5	5
Up	6	10
Left	6	6
Right	6	7
Up	7	7
Down	7	7
Right	7	8
Up	8	8
Down	8	9
Right	8	8
Left	9	9
Down	9	9
Right	9	9
Up	10	11
Left	10	10
Right	10	10
Up	11	11
Left	11	11
Right	11	12
Up	12	12
Down	12	12
Right	12	13
Up	13	13
Down	13	13
Right	13	14
Up	14	14
Down	14	14
Right	14	15
Up	15	15
Down	15	16
Right	15	22

Table A.2: Continued state-transition table of the maze FSM.

Input	Current state	Next state
Left	16	16
Down	16	17
Right	16	16
Left	17	18
Down	17	17
Right	17	17
Up	18	19
Left	18	18
Down	18	18
Up	19	19
Left	19	20
Right	19	19
Up	20	20
Left	20	21
Down	20	20
Up	21	21
Left	21	21
Down	21	21
Up	22	22
Down	22	23
Right	22	22
Left	23	23
Down	23	24
Right	23	23
Left	24	24
Down	24	25
Right	24	24
Left	25	26
Down	25	25
Right	25	25
Up	26	26
Left	26	27
Down	26	26
Up	27	27
Left	27	27
Down	27	28
Left	28	29
Down	28	28
Right	28	30
Up	29	29
Left	29	29
Down	29	29
Up	30	30
Down	30	30
Right	30	31
Up	31	31
Down	31	32
Right	31	31
Left	32	1

Appendix B

README

To create a new Anaconda environment that will install the packages used in this project, use the following command in the Anaconda prompt while being inside this directory:

conda env create -f environment.yml

Then run the following command to activate the environment:

conda activate FSM_RL

In order to use the Jupyter notebooks, run the following command in the Anaconda prompt inside this directory:

jupyter lab

And done! A local Jupyter website should show up inside your default web browser. If that does not happen, try to load the following website:

http://localhost:8888/lab

Another method is to use a Visual Studio Code IDE with the extensions for Python and Jupyter, which will create a local jupyter kernel that will use the conda environment.

The files containing the finite state machines are located inside the ./csv_files/ directory. The main Jupyter notebooks that were used during the development of this project are located inside the ./evaluation_jupyter_notebooks/ directory.

Appendix C

Record of supervision

Appendix

RECORD OF SUPERVISION

NB: This sheet must be brought to each supervision and submitted with the completed Dissertation

(to be completed as appropriate by student and supervisor at the end of each supervision session, and initialed by both as being an accurate record. NB it is the student's responsibility to arrange supervision sessions and he/she should bear in mind that staff will not be available at certain times in the summer)

Student Name: Maciej Aleks Legas
Student Number: 2031545
Dissertation Title: Using deep reinforcement learning for exploring state spaces of finite state machines
Supervisor: Dr Michael Edwards

Supervision	***Date, duration***	***Notes***	***Initials Supervisor***	***Initials student***
1: Brief outline of research question and preliminary title (**by pre June**)	28 April 2021	Project ideas discussion, a look through previous works	ME	MAL
2: Discussion of detailed plan and bibliography (**by June**)	13 May 2021	Discussion regarding the needed implementation steps	ME	MAL
3: Progress report, discussion of draft chapter (**by August**)	22 July 2021	Implementation part, discussion regarding possible RL algorithms	ME	MAL
4: (optional) progress report (**by September)**	6 September 2021	Further work on the implementation	ME	MAL
5: Submission (**by 30 September**)	29 September 2021	Continued implementation work	ME	MAL

Statement of orginality

I certify that this dissertation is my own work and that where the work of others has been used in support of arguments or discussion, full and appropriate acknowledgement has been made. I am aware of and understand the University's regulations on plagiarism and unfair practice.

Signed: ...M. Legas..... Date: 15/12/21