Prague University of Economics and Business
Faculty of Informatics and Statistics

Assessing Policy Optimization agents using Algorithmic IQ test

MASTER THESIS

Study program: Knowledge and Web Technologies
Specialization: Security Management

Author: Petr Zeman
Supervisor: Ing. Ondřej Vadinský, Ph.D.

Prague, June 2023

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Vadinský Ph.D. for his many advices and assistance with testing of my updated version of AIQ. Next, I would like to thank the Metacentrum, whose Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic. I would also like to thank the existence of the Grammarly program, which significantly helped me in proofreading my work.

Finally, I wish to give my heartfelt thanks to my family who assisted and supported me while I worked on this thesis.

Abstrakt

Tato diplomová práce nejdříve čtenáře uvede do historie a ideologie formalizovaného testování inteligence, než se pustí do principů testu Algoritmického IQ (AIQ). Jelikož má AIQ mnoho podobného s frameworkem Posilovaného Učení, je následující kapitola dedikovaná principům tohoto frameworku. Součástí této kapitoly je i představení některých populárních agentů uživateli, po něm následuje hlubší analýza agentů vybraných pro tuto diplomovou práci: Vanilla Policy Gradient a Proximal Policy Optimization.

Praktická část diplomové práce nejprve představí historii prototypové implementace testu Algoritmického IQ včetně popisu kódu. To je následováno popisem mé práce na oživení kódu testu z Python 2 na Python 3, opravami pro Operační Systém Windows, Implementací systému pro logování chyb agenta a pár dalšími menšími úpravami.

Další část této diplomové práce se soustředí na problémy při implementaci agentů založených na moderních architekturách Posilovaného Učení do testu AIQ. Jelikož AIQ test pracuje s prostředími inverzně vůči moderním standardům OpenAI Gym a jejich variacemi, bylo potřebné kód agentů transformovat do stavu schopného spolupracovat s AIQ testem.

Nakonec byly implementovány agenty testování pro nalezení vhodných výchozích hodnot nového parametru a získané hodnoty byly využity pro další testy. Skrze statistické metody byly mezi sebou porovnány výsledky nově implementovaných agentů. Jejich výsledky, díky podobnosti s dalšími testy, podporují použitelnost AIQ jako nástroje pro testování inteligence. Následné porovnání nových agentů s těmi původně implementovanými dává zajímavé informace o jejich inteligenci.

Klíčová slova

Test Algoritmického IQ, Posilované učení, Agenti typu Policy Gradient, Hodnocení Inteligence Agentů, Vanilla Policy Gradient, Proximal Policy Optimization

Abstract

This thesis briefly introduces the history and ideology behind the formalised evaluation of intelligence before focusing on the workings of Algorithmic Intelligence Quotient test. As AIQ is closely related to the Reinforcement Learning framework, the following chapter is dedicated to the principles of this framework and introduces some of the popular agents before focusing more in-depth on the agents I have chosen for implementation: Vanilla Policy Gradient and Proximal Policy Optimization.

The practical part of this thesis first introduces the history of the prototypical implementation of the Algorithmic Intelligence Quotient and a description of how its code work. This is closely followed by my update of the code base from Python 2 to Python 3, fixes for the code on the Windows operating system, the introduction of a system for logging a failure of an agent and a few other minor tweaks.

Next, the thesis focuses on the complexities in introducing agents based on modern Reinforcement Learning architectures into the AIQ test. As AIQ works with environments inverse to the modern standards of OpenAI Gym and its variations, it was necessary to transform them into a state compatible with the AIQ test.

The agents are tested to find good default values of a newly introduced parameter before using various statistical methods to compare the new agents against each other and then against originally implemented agents by using the acquired data. Due to its similarity to other existing benchmarks of implemented agents, the results of the comparison between newly implemented agents support the viability of AIQ as an evaluation tool. The results of the comparison to the initially implemented agents give interesting insight into various agents' intelligence.

Keywords

Artifficial Intelligence Quotient test, Reinforcement Learning, Policy Gradient agents, Agent Intelligence Evaluation, Vanilla Policy Gradient, Proximal Policy Optimization

Contents

List of Figures

List of Tables

List of abbreviations

AI Artificial Intelligence
AGI Artificial General Intelligence
AIQ Artificial Intelligence Quotient
UI Universal Intelligence
RL Reinforcement Learning
Policy Optimisation Agents RL agents looking for state value function
Q-Learning RL Agents looking for action-value function Q*
Policy Gradient Agents Policy optimisation agents that utilise gradient descent or ascent to maximise value
MBVE Model-Based Value Expansion
I2A Imagination-Augmented Agents
MBMF Model-Based RL with Model-Free Fine-Tuning
DQN Deep Q-Networks
DDQN Double DQN
C51 Categorical 51-Atom DQN
QR-DQN Quantile Regression DQN
HER Hindsight Experience Replay
VPG Vanilla Policy Gradient
TRPO Trust Region Policy Optimization
PPO Proximal Policy Optimization
PPO-Penalty Adaptive KL Penalty Coefficient variation of PPO
PPO-CLIP Clipped surrogate objective of PPO
A2C / A3C Asynchronous Advantage Actor-Critic
DDPG Deep Deterministic Policy Gradient
TD3 Twin Delayed DDPG
SAC Soft Actor-Critic
EGLP Expected Grad-Log-Prob Lemma
ADAM Adaptive Moment Estimation
GAE Generalised Advantage Estimate
KL-Divergence Kullback-Leibler Divergence or “relative entropy”
SPE Parameter Steps_per_Epoch
AUC Area Under Curve
MW-U Mann Whitney U test

Introduction

"*A fundamental problem in artificial intelligence is that nobody really knows what intelligence is.*"(Legg; Hutter, 2007) With this notion, Legg and Hutter enter into the overlooked area of definition and evaluation of intelligence. With a need to find an interpretation specific to matters of artificial intelligence, Leggs and Hutter analysed a great number of various formal and informal definitions to create a new one. This definition was named *Universal Intelligence*.

Along with the definition, a way to achieve a way to evaluate artificial agents is necessary. "*A fundamental problem in strong artificial intelligence is the lack of a clear and precise definition of intelligence itself.*" (Legg; Veness, 2011b) By expanding the idea of his previous work Leggs joined with doctor Veness and focused on the creation of a new exam that would allow quantifying the Universal Intelligence of tested agents. Through abstracting the original idea of Universal Intelligence, a metric called Algorithmic Intelligence Quotient was created with an acronym of AIQ. Along with this metric, an exam that automatically measures the values of AIQ was created and successfully tested on a couple of basic agents. To further test the applicability of this evaluation, this work will focus on implementing more complex artificial agents based on Policy and Policy Gradient Optimization into the AIQ test and compare the achieved results to the original results.

The goal of this work can be therefore defined in these steps:

- Introduce Reinforcement Learning framework and Policy Optimization agents.
- Review the principles of general intelligence tests for artificial agents with focus on Algorithmic Intelligence Quotient test.
- Implement the chosen Policy Optimization agents (Vanilla Policy Gradient, Proximal Policy Optimization) into the AIQ test.
- Evaluate the implemented agents using the AIQ test and compare them to the original agents.

The first two steps are fulfilled through extended research that begins by presenting the principles of both formal and informal evaluation of intelligence in chapter 1 with a greater focus on formal definitions, especially Universal Intelligence and Algorithmic Intelligence Quotient. Reinforcement Learning is introduced in chapter 2 that introduces the basics behind this category of AI agents before introducing various existing agents with their basic ideas, before diving more in-depth to agents chosen for implementation in this thesis: Vanilla Policy Gradient and Proximal Policy Optimisation.

Before focusing on the implementation of new agents into the AIQ test a small update of base code is required as the original AIQ test is still running on Python 2. As this version of Python has long since been deprecated, chapter 3 describes the original version of AIQ and the process done to upgrade it to newer versions of Python 3, along with the introduction of

further improvements to this evaluation program. After achieving a more modern compatible version of the AIQ test, chapter 4 focuses on the complexities of implementation of chosen Policy Optimisation agents Vanilla Policy Gradient and Proximal Policy Optimisation into the AIQ test. This chapter introduces various Python packages that implement chosen agents before looking into why it is very difficult to use these packages without any modification. This is followed by a description of the architecture of a package chosen for modification before introducing the new agents' architecture using parts of the code from the package described.

The final chapter 5 focuses on the last goal of the thesis in the evaluation of the implemented agents using the AIQ test. As the AIQ test is very computationally expensive, the focus of this thesis was on tests over a default configuration of agents. This was further complicated by the existence of a new hyperparameter in our agents lacking a default parameter value. Instead, an experiment was created that searched for good configurations of this parameter on default configurations of other existing hyperparameters using an interpolation of acquired data. Data acquired this way was then utilised to first statistically compare Vanilla Policy Gradient and Proximal Policy Optimisation against each other for confirmation that the results acquired are similar to existing benchmarks for further support of the correct implementation of agents and AIQ test's validity in evaluating intelligence. After this comparison the final analysis utilised this data to compare newly implemented agents to the original.

1. Evaluation of Intelligence

Evaluating intelligence is a difficult task. What is intelligence? "Perhaps the ability to learn quickly is central to intelligence? Or perhaps the total sum of one's knowledge is more important? Perhaps communication and the ability to use language play a central role? What about "thinking" or the ability to perform abstract reasoning? How about the ability to be creative and solve problems? Intelligence involves a perplexing mixture of concepts, many of which are equally difficult to define."(Legg; Hutter, 2007)

Because intelligence is such a complex concept, one cannot evaluate it directly. First, one has to define what exactly it is that one wishes to assess. This definition can take the form of either a formal or informal definition. Informal definition purely defines into words a specific part of intelligence that is the focus of the evaluation. The formal definition further expands on the informal definition by transforming such a definition into a quantifiable variant, usually in the form of an equation.

This chapter begins with an introduction to the different categories of artificial intelligence. After this section, the historically important tests for general intelligence will be separated into those using informal definitions and those using formal ones. While a brief overview of the first group of definitions and their history will be done in section 1.2, this thesis focuses on the second group. Formalisation of a definition brings significant advantages that span beyond the academic sphere. It allows to transfer same information better no matter the current context. Why this is important is briefly introduced at the beginning of section 1.3. After explaining the advantages of formalised definitions, some relevant histories of formal definitions of intelligence are introduced. The final section of this chapter is dedicated to the theory behind Algorithmic Intelligence Quotient that is utilised by this thesis

1.1 Categorization of Artificial Intelligence

Artificial intelligence has many different views on how it should be categorised. and some definitions do not have one specific interpretation. To better understand this thesis's focus, let us briefly examine the different views on AI.

1.1.1 Strong and Weak AI

One of the main categorizations of AI is between "strong" and "weak" AI introduced in the Chinese Room thought experiment (Searle, 1980). "*According to weak AI the principal value in the study of the mind is that it gives us powerful tool*"(Searle, 1980) As can be seen, the goal of weak AI is not understanding thinking but on the applicability of AI for work. On the other hand, in the strong AI "*the computer is not merely a tool in the study of the mind;*

rather, the appropriately programmed computer really is a mind"(Searle, 1980). The focus is on explaining how cognition, both human and artificial, works and to one day create a machine as capable of reasoning as a human.

1.1.2 Narrow and General AI

A modern understanding of AI is in many ways similar to some interpretations of strong and weak AI. While what is called "narrow" and "general" AI does not have any specific definition and any understanding is still being actively researched, (Goertzel, 2015) utilised some previous works to explain the difference between these two types of AI. "*For a narrow AI system, if one changes the context or the behaviour specification even a little bit, some level of human reprogramming or reconfiguration is generally necessary to enable the system to retain its level of intelligence. Qualitatively, this seems quite different from natural generally intelligent systems like humans, which have a broad capability to self-adapt to changes in their goals or circumstances, performing "transfer learning" to generalise knowledge from one goal or context to others.*" A possible interpretation of this sentence can be that narrow AI focuses on achieving high ability of a very specific or "narrow" types of tasks but is lost if applied to anything else. General AI, on the other hand, tries to achieve more universal intelligence that is not only capable of working in various fields but is also capable of transferring experiences achieved from solving problems in one field to another. This transfer of information between various fields is otherwise known as "learning" and is the desired goal of general AI.

As previously stated, AGI has no specific definition and many different approaches to its understanding. To dive deeper into this rabbit hole would mean explaining many different concepts of intelligence, which is different from the focus of this thesis. This thesis utilises the "*Mathematical Approach*" of understanding General Intelligence typified by (Legg; Hutter, 2007). This understanding believes that "*Truly, absolutely general intelligence would only be achievable given infinite computational ability. For any computable system, there will be some contexts and goals for which it is not very intelligent. However, some finite computational systems will be more generally intelligent than others, and it is possible to quantify this extent*"(Goertzel, 2015). One possible way of quantification is further explained in 1.3.3.
For other approaches, refer to the introduction to AGI in (Goertzel, 2015).

1.2 Tests Utilising Informal Definitions

Informal definitions of general intelligence mostly try to compare machine intelligence to aspects of humans, focusing on specific parts of intelligence any such evaluation is assessing. Informal definitions are the original way of defining intelligence. It could be said that one of the first pioneers into the idea of general intelligence was René Descartes in the 17th century. In the 5th part of his work (DESCARTES, 1637), Descartes raises a question of what is unique to humans compared to animals. Two inherent abilities were defined. The ability to

be universal in thinking and the ability to speak comprehensively. He postulated that while machines or animals may be able to make sounds similar to words or even sentences, the ability to flexibly use those sentences to create fully coherent and meaningful sentences other than the ones they learned is unique to human intelligence. This postulation is centuries later used as a basis for one of the most important tests of general intelligence - the Turing test.

1.2.1 Turing Test based tests

(Alan M Turing et al., 1950) raises the idea of machines being capable of thinking and a question asking if an average human can differentiate between a human and a machine. This question is formed into a test for machines based on an "imitation game". In this test, a human and a machine pass sentences to a human interrogator in another room. This interrogator then has to decide if the passed sentence was written by a human or a machine. Machine is defined as intelligent when a difference between the results of a human and a machine cannot be found. While a novel and very popular idea, this test still had many problems that were noticed during the following decades.

- (Searle, 1980) proposes that the Turing test focuses too little on the idea of understanding and that it is possible to create machines capable of behaving humanely just by imitating specific actions and not truly understanding what they are doing. This is explained in his theory of *Chinese room*.
- (Harnad, 1991) raises the problem of sidelining other types of communication than language. To amend this problem, the *Total Turing Test* is proposed. In this test, the robot interacts with its surroundings directly. This makes the target of the test not only the language capabilities of the robot but also all other intelligent behaviour.
- (Schweizer, 2012) thinks that the focus of the test should be on species instead of individuals. Language is a social construct that varies significantly between different species. To know if machines or other cognitive types can genuinely think, it is not enough to see them parasite upon intelligent behaviour and existing language but to see if they can create their own behaviour and language. Consequently, the target of *The Truly Total Turing Test* is to find out if the evaluated cognitive type can create its own intelligent behaviour and language.

To achieve an ideal informal definition was the goal of last centuries. However, talking about those centuries in depth is not the goal of this thesis, and for more in-depth information, refer to the overview in work (Vadinský, 2019).

1.3 Tests Utilising Formal Definitions

Heylinghen explores the advantages and disadvantages of formal expressions. (Heylighen, 1999) states that three major advantages are:

- **Long-term knowledge storage** - Formal expressions allow for knowledge to be understood even if context changes
- **Capacity for universal communication** - Through formalised expression, meaning can be understood without the need for specification of the context. This allows for sharing messages across a wider variety of people.
- **Testability of formalised knowledge** - Unless the test is formulated to be context-independent, the results may have little to do with the original proposition. Formalisation allows for this context-independence.

Of the three advantages, the most important advantage for testing general intelligence is the advantage of testability. Utilising formalised definitions of intelligence increases the precision and objectivity of intelligence evaluations and enables automatic evaluation of intelligence.

1.3.1 Early History of Formal Definitions of Intelligence

Historically, there was a tendency in the AI community to contrast artificial intelligence with human intelligence, an action that merely passed the buck to psychologists. Then John McCarthy released his article *What is artificial Intelligence* (McCarthy, 1998) in 1998 (later revised in 2003, 2004 and 2007), where the following question was asked:
"*Q. Isn't there a solid definition of intelligence that doesn't depend on relating it to human intelligence?*

A. Not yet. The problem is that we cannot yet characterise in general what kinds of computational procedures we want to call intelligent. We understand some of the mechanisms of intelligence and not others."

While this difficulty redirected much research effort from artificial general intelligence to artificial narrow intelligence, a new research direction was formed.

Some of the earliest attempts to formalise intelligence were focused on a similar set of ideas as Turing's test and its derivatives - language. Some notable proposed ways to test intelligence through language were, for example, compression tests (Mahoney, 1999) proposing to use compression ratio of text compression for evaluation of AI and Linguistic complexity proposed by HAL project (Treister-Goren et al., 2000) to evaluate intelligence through metrics like vocabulary size, degree of syntactic complexity and many others.

Another branch of research focused on the idea that intelligence is "*The ability to deal with complexity*" (Gottfredson, 1997) and that the most important questions in testing intelligence are the most complex ones. While this idea is difficult to implement in practice, it was chosen as a base behind two important paths of formalisation intelligence. C-Tests (Hernandez-Orallo, 1999) and Universal Intelligence (Legg; Hutter, 2007). These two paths were later expanded in the Anytime Intelligence Test, which combined ideas of C-tests and Universal Intelligence along with inspirations from other sources to create a formal intelligence test. Ideas from all of these then crystallised in Algorithmic Intelligence Quotient (AIQ) test (Legg;

Veness, 2011b) that created both an approximation of Universal Intelligence and a prototype of a test to compute this approximation.

As this text focuses on Universal Intelligence, the following chapters will focus mainly on the second research branch. We will first look into C-tests and Universal Intelligence. Afterward, the Anytime intelligence test will be briefly introduced before leaving the theory behind AIQ for an entirely new section in 1.4.

1.3.2 C-Tests

In 1999, Hernandez-Orallo proposed the definition of intelligence as the "ability to comprehend" and formalises this ability with the help of constructs based on descriptional complexity. According to (Hernandez-Orallo, 1999), a scientific measure of intelligence must comply with five requirements.

- **Non-boolean** - Intelligence is not an absolute attribute and cannot be defined with simple true or false evaluation.
- **Factorial** - As intelligence is multidimensional, it is necessary to account for as many dimensions or factors as possible and not measure a single intelligence factor.
- **Non-anthropomorphic** - All references to intelligent behaviour so far were dependent on human intelligence. It is necessary to create a measure of intelligence that does not depend on relation to human intelligence.
- **Computationally based** - It is necessary to be able "*to give the specification of the problem in computational terms, in order to solve the problem with AI means, which are exclusively machines and programs*". (Hernandez-Orallo, 1999)
- **Meaningful** - Intelligence must not be defined as that what is measured by intelligence tests. All measures of intelligence need to be expressed from their original definition - the ability to comprehend.

(Hernandez-Orallo, 1999) focuses on the notion that the Turing Test is merely a test of humanity instead of intelligence and compares it to Comprehensive tests. Comprehensive tests or C-tests were created to assess the language proficiency of non-native speakers, and Hernandez-Orallo proposes that these tests can also be employed for analysing the comprehensive ability of machines. These tests work by taking a text that has parts removed. The tested agent is then made to choose from a list of choices to fill in missing parts. This list contains not only correct choices but also some additional choices as distractors. The result is then measured through an equation that computes a weighted sum of the number of correct answers for each test case, where the weight is proportional to the difficulty of the test case (as measured by exponent e). The resulting value provides a measure of the overall performance of the intelligent system on the test set. While C-tests measure only one factor defined here as *fluid intelligence* (Hernandez-Orallo, 1999) also proposes that other independent factors can be measured through certain extensions like *Knowledge Applicability*, *Contextualisation* or *Knowledge Construction*.

1.3.3 Universal Intelligence

In their article, Legg and Hutter begin with a basic question - what is intelligence:
"*Perhaps the ability to learn quickly is central to intelligence? Or perhaps the total sum of one's knowledge is more important? Perhaps communication and the ability to use language play a central role? What about "thinking" or the ability to perform abstract reasoning? How about the ability to be creative and solve problems? Intelligence involves a perplexing mixture of concepts, many of which are equally difficult to define.*" (Legg; Hutter, 2007)
To fix the problem of needing more concrete and precise definitions of intelligence, Shane Legg and Marcus Hutter dive into various sources to create a new formal definition for intelligence. Through scrutinising dozens of informal definitions of intelligence, researching possible categories of agents and environments, and acquiring inspiration from few existing formal definitions (Legg; Hutter, 2007) creates first an informal definition that is then mathematically formalised into an equation that should define a general measure of intelligence for arbitrary machines and other agents.
The new informal definition is defined in the following sentence:

Informal Definition:. *Intelligence measures an agent's ability to achieve goals in a wide range of environments. (Legg; Hutter, 2007)*

To formalise this definition, we must first specify this sentence's main parts. They are agent π, environment μ, and goals expressed for an agent by reward value from *reward space* $\mathcal{R}$. The interaction of these parts makes it possible to approximate the agent's ability. The interaction behaves followingly: The agent has a set of symbols that he can send to the environment. This set is called *action space* and denoted as $\mathcal{A}$. The environment has a similar set of symbols from a finite set called the *perception space* denoted as $\mathcal{P}$ that he uses to transfer information back to the agent. The last set is called *reward space*. The agent and environment take turns sending information to each other through symbols. Environment sends both *perception* symbol and *reward* for the last action simultaneously. In response, the agent's acting function π takes the current history as input and chooses his next *action* symbol to pass to the environment. The agent here can be practically anything. The environment can be defined as the probability that specific observation and reward happen given the current interaction history between agent and environment.(Legg; Hutter, 2007).

So far, aside from some differences in naming, this framework is identical to Reinforcement Learning used in Artificial Intelligence. This relevancy will be important later in the thesis, and for more information about Reinforcement Learning, refer to 2. Nevertheless, a framework is only the base template, and some problems must be fixed to create a measurable definition of intelligence.

Measure of success - how to formalise the idea of "success" for an agent in both the short and long term. Usually, this is achieved by discounting rewards so that they decay geometrically into the future. Unfortunately, this is complicated to implement sufficiently. However, by requiring environments never to exceed the reward value of 1, the sum of all

rewards becomes definitely finite. These *reward summable environments* μ then already contain temporal factor, and discount is no longer required. and so the reward sum V_μ^π is always bounded (Legg; Hutter, 2007). However, for a truly perfect expected value of an agent, it would be necessary to sum infinite rewards of agents in environments which makes this incomputable

Space of environments - how to sufficiently achieve and test on "wide range of environments" If environments have no restrictions, they cannot be described finitely. Environments that cannot be described finitely are incomputable, and testing agents on an incomputable environment with a computer is impossible. As such, one of the requirements for environments is that it has to be computable. Computability is achieved by generating environments through instructions in a prefix universal Turing machine $\mathcal{U}$ called *reference machine*. Another thing to test agents on is how well they can both approach the principle of Occam's Razor [1] and how the agent can deal with complexity. As such, a wide variety of complexity is required for a set of environments used for testing agents, along with a weighted reward model for each environment based on how complex such an environment is. Implementing this requires an ability to measure the complexity of any environment. (Legg; Hutter, 2007) proposes a Kolmogorov complexity [2] due to its near independence on a reference machine. To approximate Kolmogorov complexity, a simple encoding method of expressing indexes as binary string $< i >$ is utilised. With this complexity of environment μ can be defined as $K(\mu) := K(< i >)$. In order to formalise Occam's razor, it is also necessary to assign a probability to environments to make complex environments more likely and simple ones less likely. Thanks to the previous definition of each environment being defined by a binary string, "it is possible to reduce the environment's probability by one half for each bit of program, reflecting the fact that each bit has two possible states. This gives us what is known as the *algorithmic probability distribution* over the space of environments defined $2^{-K(\mu)}$)

After applying these solutions, we achieve the following mathematical formalisation:

Formal Definition:. *Let E be the space of all computable reward summable environmental measures with respect to the reference machine $\mathcal{U}$ and let $\mathcal{K}$ be the Kolmogorov complexity function. The expected performance of agent π with respect to the universal distribution $2^{-\mathcal{K}(\mu)}$ over the space of all environments E is given by:*

$$\Upsilon(\pi) := \sum_{\mu \in E} 2^{-K(\mu)} V_\mu^\pi$$

We call this the universal intelligence of agent π (Legg; Hutter, 2007)

In this equation, $\Upsilon(\pi)$ defines the Universal Intelligence measure of agent pi that is computed

[1] **Occam's Razor**: "*Given multiple hypotheses that are consistent with the data, the simplest should be preferred.*"

[2] Kolmogorov complexity defined in (Kolmogorov, 1965) is a theoretical measure of the amount of information in an object. An object's $\mathcal{K}$ complexity is the length of the shortest possible program or algorithm that can generate that object. However, no algorithm exists that could calculate this length finitely, so this complexity is uncomputable.

from the sum of *algorithmic probability distribution* over space of environments $2^{-K(\mu)}$ and capability of what is an agent capable of achieving in Value function V_μ^π

This definition still has many problems. Utilising Kolmogorov's complexity which is uncomputable, means that even the value of Υ is uncomputable. Additionally, the test itself works with infinitely many environments, which creates problems when trying to create a practical test. Even through these problems, *Universal Intelligence* created a fundamental building block upon which future formal tests of machine intelligence were built thanks to the many positives Universal Intelligence has. (Legg; Hutter, 2007) also defined the desirable properties of intelligence measure and how Universal Intelligence these measures fulfills in chapters 3.5 and 4.3. The following list is created from the information contained in those chapters.

- **Valid** - ". A test/measure of intelligence should be just that, it should capture intelligence and not some related quantity or only a part of intelligence" (Legg; Hutter, 2007)
  - This was achieved through step-by-step formalising mainstream informal definitions so as long as the informal definition defined by Universal Intelligence holds, the result can be defined as formal definition of intelligence.
- **Informative** - "The result should be a scalar value, or perhaps a vector, depending on our view of intelligence. We would like an absolute measure of intelligence so that comparisons across many agents can easily be made." (Legg; Hutter, 2007)
  - Universal intelligence has a real value to be used to compare different agents
- **Wide range** - "A test/definition should cover very low levels of intelligence right up to super human intelligence" (Legg; Hutter, 2007)
  - Universal intelligent is able to test order agents from the most basic to the theoretical super intelligent agent AIXI[3]
- **General** - "Ideally we would like to have a very general test/definition that could be applied to everything from a fly to a machine learning algorithm"(Legg; Hutter, 2007)
  - Universal intelligence factors agent's performance on all well defined environments into its value of Universal Intelligence measure
- **Dynamic** - "A test/definition should directly take into account the ability to learn and adapt over time as this is an important aspect of intelligence" (Legg; Hutter, 2007)
  - Universal intelligence achieves this through its implementation of reward summable environment μ;
- **Unbiased** - " A test/definition should not be biased towards any particular culture,

[3]One of the contributors behind the genesis of definition of Universal Inteligence is the theoretical optimal agent AIXI as defined in (Hutter, 2004). Part of AIXI was even used to derive Universal Intelligence - more specifically the *Ingelligence Order relation*. Explained simply, the intelligence order relation in AIXI is a partial ordering of agents based on their relative abilities to achieve their goals in different environments. AIXI is defined as the most intelligent agent, with it being universally optimal. With AIXI being theoretically universally optimal, an upper limit of Universal intelligence can therefore be defined as the intelligence of AIXI. Sadly, AIXI is only theoretical and cannot be practically implemented due to the uncomputability of *Kolmogorov's complexity*. Additionally, the notion of universal optimality of AIXI was argumented against in (Leike et al., 2015). This paper brought forward multiple arguments for the power of AIXI being extremely relative to the choice of the prefix Universal Turing Machine.

species, etc" (Legg; Hutter, 2007)
  - Universal Intelligence measures tries to achieve this property through something very general and principled - Universal Turing Computation - because concept of computation appears to be a fundamental theoretical property instead of being a product of any specific culture. By weighing different environments depending on their Kolmogorov complexity and space of all computable environments, the Universal Intelligence measure mostly avoids the definition of intelligence without respect to any particular culture, species, etc. However, it has not entirely removed the problem because environmental distribution depends on the reference machine. A partial fix is proposed by using the simplest possible reference machine.
- **Fundamental** - "We do not want a test/definition that needs to be changed from time to time due to changing technology and knowledge" (Legg; Hutter, 2007)
  - Universal Intelligence measure is based on computation, information, and complexity.
- **Formal** -"The test/definition should be specified with the highest degree of precision possible, allowing no room for misinterpretation. Ideally, it should be described using formal mathematics" (Legg; Hutter, 2007)
  - Universal Intelligence has a formal definition in the form of an equation.
- **Objective** - "The test/definition should not appeal to subjective assessments such as the opinions of human judges." (Legg; Hutter, 2007)
  - "Universal Intelligence does not depend on any subjective criteria" (Legg; Hutter, 2007)
- **Fully Defined** - "Has the test/definition been fully defined, or are parts still unspecified?" (Legg; Hutter, 2007)
  - The Universal Intelligence has been derived and fully defined in (Legg; Hutter, 2007)
- **Universal** - "Is the test/definition universal, or is it anthropocentric?"(Legg; Hutter, 2007)
  - "Universal Intelligence is in no way anthropocentric" (Legg; Hutter, 2007)
- **Practical** - "A test should be able to be performed quickly and automatically, while from a definition it should be possible to create an efficient test" (Legg; Hutter, 2007)
  - Universal Intelligence measure in the original definition cannot be turned into a test of intelligence due to the uncomputability of Kolmogorov complexity.

These principles are then applied to some existing tests with the following results:

Intelligence Test	Valid	Informative	Wide Range	General	Dynamic	Unbiased	Fundamental	Formal	Objective	Fully Defined	Universal	Practical	Test vs. Def.
Turing Test	•	·	·	·	•	·	·	·	·	•	·	•	T
Total Turing Test	•	·	·	·	•	·	·	·	·	•	·	·	T
Inverted Turing Test	•	•	·	·	•	·	·	·	·	•	·	•	T
Toddler Turing Test	•	·	·	·	•	·	·	·	·	·	·	•	T
Linguistic Complexity	•	●	•	·	·	·	·	•	•	·	•	•	T
Text Compression Test	•	●	●	•	·	•	•	●	●	●	•	●	T
Turing Ratio	•	●	●	●	?	?	?	?	?	·	?	?	T/D
Psychometric AI	●	●	•	●	?	•	·	•	•	•	·	•	T/D
Smith's Test	•	●	●	•	·	?	●	●	●	·	?	•	T/D
C-Test	•	●	●	•	·	●	●	●	●	●	●	●	T/D
Universal Intelligence	●	●	●	●	●	●	●	●	●	●	●	·	D

Table 1: In the table ● means "yes", • means "debatable", · means "no", and ? means unknown. When something is rated as unknown that is usually because the test in question is not sufficiently specified.

Test vs. Def. Finally, we note whether the proposal is more of a test, more of a definition, or something in between.

Figure 1.1: Universal Intelligence - Intelligence test principles (Legg; Hutter, 2007)

From this comparison, it can be seen that Universal Intelligence has many positives and is worth further analysis. That UI is worth further analysis is also proven by, for example, the work of (Hibbard, 2009). This work dissects the theory of Universal Intelligence to prove that certain choices of prefix Universal Turing Machine can bring extreme bias to the formal measure of intelligence and that even a simple constraint on programs defining environments like minimum length limit can help fix this. Additionally, Hibbard refers to the current understanding of physics and No Free Lunch Theorem[4] to prove that it is not unreasonable to have purely finite environments and that there is a necessity to have unequal weighting of environments such as the Kolmogorov complexity.

As the Universal Intelligence measure is still only a definition of intelligence and cannot be used as a test, it is necessary to create one. Creating one is impossible for the original definition of the UI measure due to the incomputability of $\mathcal{K}$ and V_μ^π. Instead, (Legg; Veness, 2011b) creates an approximation of UI they call AIQ, allowing them to create a practical test prototype that measures this approximation. We will return to AIQ 1.4. Before that, it is necessary to look at the Anytime Intelligence test.

[4](Hibbard, 2009) refers to No Free Lunch Theorem as specified by (Wolpert et al., 1997). This theorem says "that all optimisation algorithms have equal performance when averaged over all finite problems."

1.3.4 Anytime Intelligence test

Another formal test of intelligence is the Anytime Intelligence test proposed in (Hernández-Orallo et al., 2010). This test is based upon their previous work in revisiting C-Tests, extending knowledge from *Universal Intelligence* (Legg; Hutter, 2007) and ideas from proposed Inductive Learning extension to Turing test (Dowe et al., 1998)
(Hernández-Orallo et al., 2010) named this test Anytime Intelligence because it must be able to stop anytime and successfully give an approximation of the tested agent's Universal Intelligence. Any further testing beyond the point of stopping would only serve to achieve a more precise approximation of Universal Intelligence. Aside from this, the test proposes ways to fix other problems of the Universal Intelligence measure.

The proposed solution of (Hernandez-Orallo, 1999) for problems causing uncomputability of Universal Intelligence measure:

- Total reward of UI environments is uncomputable due to testing over infinite environments - Uncomputable amount of both interactions and environments
  - Replacing an infinite amount of environments for a sample of environments that are very sensitive to the agent's actions. *"More precisely, we want an agent to be able to influence rewards at any point in any subenvironment"*(Hernández-Orallo et al., 2010) This allows to both approximate the infinite set of environments and also to filter the environments where agent's actions would have minimal or not even any influence of its surroundings.
  - Utilising a limited amount of interactions between agents and environments while averaging reward by the number of interactions for the final score. "Interactions are not infinite. Rewards are averaged by the number of actions instead of accumulated. This makes the score expectation less dependent on the available test time." (Hernández-Orallo et al., 2010)
- Complexity of Environment is uncomputable due to Kolmogorov's complexity being uncomputable
  - (Hernández-Orallo et al., 2010) proposes replacing uncomputable Kolmogorov's complexity with their own variation of Levin's K_t complexity function. This variation is limited through both time and the number of interactions with the environment.

(Hernández-Orallo et al., 2010) also proposed additional improvements upon the original definition of UI measure:

- Time is ignored in the original definition of UI measure. This test includes a fixed time of interacting for a single agent, which grows progressively over time to achieve the "anytime" capability of the test.
  - It is necessary to avoid random but fast agents testing better than truly thinking agents - to fix this, rewards and penalties are included in environments by changing the reward range to go from -1 to 1. Additionally, requiring balanced environments

allows for agents behaving randomly to score 0.

- It is possible to use some very slow environments for testing UI measure
  - Additional constraint is given to a sample of environments. To avoid slow environments, a sample is required to contain mainly reward-sensitive environments.

Through these proposals, (Hernández-Orallo et al., 2010) proposed a few different tests for different capabilities of agents. Sadly, none of these tests were practically implemented. A simplified prototype was created by (Insa-Cabrera et al., 2011), but as noted by (Vadinský, 2018a), it lacks both time scale which is a crucial aspect of the AnyTime test and environments are not generated by Turing complete program, and are wholly observable by agent leading to only a subset of environments considered by UI definition being testable. Therefore, while the AnyTime test has a more robust definition than the UI measure, it still lacks practical tests that fully incorporate their full potential. On the other hand, the Universal Intelligence measure has a practical implementation in the Algorithmic Intelligence Quotient test.

1.4 Algorithmic Intelligence Quotient

So far, we have analysed some of the main predecessors of the Algorithmic Intelligence Quotient(AIQ) mainly the Universal Intelligence in section 1.3.3. Now the time has come to look into the test this work is based on - the prototype implementation of AIQ evaluation. AIQ test focuses on creating a practical approximation of UI measure that is practically testable.

The subsection 1.4.1 will look into the principles behind AIQ and the steps done to achieve an approximation of Universal Intelligence that is computable, along with the introduction of improvements done by Vadinský in his works (Ondřej, 2018), (Ondřej, 2018), (Vadinský, 2019). Basic principles of AIQ approximation will be followed by description of chosen Reference machine BF-Code in subsection 1.4.2 Along with this approximation, a prototypical implementation was created and will be analysed in later chapter 3, specifically in section 3.1 followed by a description of steps taken to improve the prototypical code in section 3.3

1.4.1 Principles of AIQ test

"*The aim of the Universal Intelligence Measure was to define intelligence in the most general, precise and succinct way possible. While these goals were achieved, this came at the price of asymptotic computability*"(Legg; Veness, 2011b). To allow for any practical use of the Universal Intelligence an approximation had to be be made. The original Universal Intelligence is approximated through the following changes:

- **Environment sampling** - Occam's razor
  - Problem: While generating samples of environments, it is necessary to both include the idea of Occam's razor and to avoid the incomputability of Kolmogorov's complexity K.
  - Solution: By utilising Solomonoff's Universal distribution's (Solomonoff, 1964) definition of machine M[5] as an approximation of Kolmogorov's complexity, the difficulty of sampling is majorly eased, even if it allows for multiple programs defining the same environment.
- **Environment simulation** - Halting problem
  - Problem: Every sampled program must be run on a defined reference Machine. Some programs will never halt and due to the halting problem proven by (Alan M. Turing, 1936) there is no process that can always determine specific cases. This results in the process getting stuck.
  - Solution: Practically there is no difference if program never halts or halts af-

[5]Solomonoff's Universal distribution assigns a probability to bit sequences that begin with the final sequence of x calculated by program p on reference Turing Machine $\mathcal{U}$ using the following equation

$$M_{\mathcal{U}}(x) := \sum_{p:\mathcal{U}(p)=x*} 2^{-l(p)}$$

(Solomonoff, 1964)

ter way too many steps. This problem is reduced through choosing a reference machine where non-halting programs are relatively unlikely or easier to detect. Any leftover problematic programs can be detected through implementing a limit of computational steps. Any programs that pass this limit in any cycle will be discarded

- **Temporal Preference** - Undiscounted bounded trials over fixed length trials
  - Problem: There is no practical way of knowing if a program will respect given reward bounds of Universal Intelligence. Universal Intelligence also asks for infinite interaction sequence for the testing.
  - Solution: While implementation of *geometric discounting* that would fix out of bound rewards is proposed, it is computationally inefficient over longer programs. In the end an alternative was chosen through focusing on undiscounted, bounded rewards over fixed length trials along with replacing the sum of rewards with an arithmetic mean. Inclusion of fixed length also creates an approximation of infinite interaction sequence fixing the second problem.
- **Reference Machines selection**
  - Problem: While more complex reference machines would allow better simulation of real world, they are far more difficult to develop.
  - Solution: For the AIQ test prototype the focus is on a very simple reference machine, ideally one where all programs are syntactically valid and there is unique end of program symbol. The BF language was chosen and modified for this case. More about this reference machine can be found in subsection 1.4.2

Aside from creating a computable approximation of UI, some steps were also made to improve the computing efficiency and lower the variance of AIQ test.

- **Variance reduction**
  - Problem: AIQ test uses Monte-Carlo sampling for obtaining accurate estimates of agent's AIQ score. This leads to estimation being very time consuming.
  - Solution: There are multiple techniques utilised to lower the complexity of Monte-Carlo sampling.
    * Exploiting the parallel nature of Monte Carlo to run the test on multiple cores
    * Utilisation of stratified sampling to group together similar types of environment prior to the AIQ score calculation to ensure testing across all kinds of environments while also focusing on those environments that have bigger variance of results leading to more interesting results.
    * Instead of estimating the AIQ score of two agents from independent samples, an estimation of difference from single set of program samples is done. This is called *common random numbers* technique.
    * The final variance reduction technique used was antithetic varieties. Instead of using one samples, two samples are used in a way that directly oppose each other. In AIQ test implementation this is done through running program twice, once with positive and once with negative rewards.

After applying these improvements the new approximation can be defined in the following equation

Algorithmic Intelligence Quotient.

$$\hat{\Upsilon}(\pi) := \frac{1}{N}\sum_{i=1}^{N} \hat{V}_{p_i}^{\pi} \quad , where \quad \hat{V}_{p_i}^{\pi} := \frac{1}{k}\sum_{i=1}^{k} r_i$$

Where $\hat{\Upsilon}$ is the AIQ approximation of universal intelligence, π is the tested agent tested over number of programs N belonging into finite set of samples S based on environments p created by generation of bits until the end of program. $\hat{V}_{p_i}^{\pi}$ here defines the measure of success in environment program. This measure is calculated as average reward achieved by agent π during a single run of program averaged by number of total interactions k

All of these variance reduction changes led to a significant performance improvement and massively lowered computational requirements compared to the original single-thread Monte Carlo implementation.

Further Improvements

AIQ test was further improved upon through the work of (Ondřej, 2018). Through reproducing the AIQ test results Vadinský realised that AIQ still contains some weaknesses and proposed some ways to ameliorate then:

- Reducing Dependence on Reference Machine
  - Introduced option to define minimal program length to the program sampler
- Decreasing high computational requirements
  - Introduced option to save AIQ score after every thousand interactions
- Testing agents with large configuration space

In his later works like (Vadinský, 2018a) some additional improvements were introduced.

- Improvements upon sanitising of programs from nonsensical, ineffective or non-discriminatory programs. Programs that do not assist in measuring agent's AIQ score.

These improvements were focused to improve upon three areas. Increasing discriminatory power in samples, achieving more effective testing practices and allowing to setup minimal length of programs for lowering dependence on reference machine.

Aside from these some other changes have been made like improving wrapper for MC-AIXI or testing Multi-Round EL Convergence Optimization on AIQ test, however these do not bring anything for the base workings of AIQ testing and will not be focused on in this work.

1.4.2 BF Reference Machine

"*One important property of a reference machine is that it should be easy to sample from. The easiest languages are ones where all programs are syntactically valid and there is a unique end of program symbol*" (Legg; Veness, 2011b)

The reference machine chosen was Urban Muller's BF language (Müller, 1993). As a simple low-level language with only 8 instructions and very small compiler (240 bytes) it already almost simulated Turing's machine instructions themselves. The instructions specify how reference machine interacts with input and output tape that is used for interactions between agent and an environment and with work tape that is used to translate data from input to output tape. "*The agent's action information is placed on input tape cells, then the program is run, and the reward and observation information is collected from the output tape. Reward is the first symbol on the output tape and is normalised to the range* -100 *to* $+100$*. The following symbol is the observation. All symbols on the input, output and work tapes are integers, with a modulo applied to deal with under/overflow conditions.*"(Legg; Veness, 2011b)

These are the 8 instructions and their interactions inside of AIQ test (Ondřej, 2018):

- $+$ - Incrementation of symbol on work tape
- **-** - Decrementation of symbol on work tape
- **,** - Reads currently selected symbol from input tape, writes it on work tape and moves input tape by a cell
- **.** - Writes currently selected symbol from work tape on currently selected cell of output tape and moves output tape by a cell
- $<$ - Moves work tape to a cell on left side of currently selected cell
- $>$ - Moves work tape to a cell on right side of currently selected cell
- [- Begins a loop if currently selected cell of work tape is not zero
-] - Defines an end of a loop

Some changes had to be made to not have environments always deterministic. This was achieved by adding a new symbol "%" that writes a random symbol into currently selected position of input tape

Final change of the BF code instruction set was a specific instruction "#" that signifies the end of the program. This instruction allows us to know when to stop sampling on $M_{\mathcal{U}}$.

The original reference machine was further improved by defining a step limit and having history of previous agent actions placed on the input cell to make it easier for environment to compute functions of the agent's past actions. Finally the code is sanitised by removing some pointless code combinations for more compact programs almost without infinite loops. Additionally all programs lacking any instructions to read from input or write to output were also removed

2. Reinforcement Learning

One of the most basic and main goals of AI development is to achieve a state where a developed agent can expand its own knowledge and capabilities. In other words, to allow Artificial Intelligence agent to "learn" like a human. But what does it mean to learn like a human?

"*When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves.*"(R. S. Sutton et al., 2018)

2.1 Introduction

As described in (R. S. Sutton et al., 2018), Reinforcement learning is based on trial-and-error learning. The first investigation into a computational approach to this type of learning dates back to 1954, when two separate groups, Minsky and the duo of Farley and Clark, proposed neural networks designed to achieve this type of learning. Later in the 1960s, the term "*Reinforcement Learning*" was first utilised in the engineering literature. During the following sixty years, this area of Artificial Intelligence achieved significant progress and became one of the strongest players in AGI. To learn more about the advances during this time frame, the (R. S. Sutton et al., 2018) is highly recommended.

Reinforcement Learning introduced a computational approach to learning from the agent's interactions with its surroundings. An agent is never told what he is supposed to do, but its every action is either rewarded or punished. Through trial-and-error agent must find out and remember which actions are best not only immediately, but even during longer time frame. This is achieved through constant modification of policies that define how agent behaves.

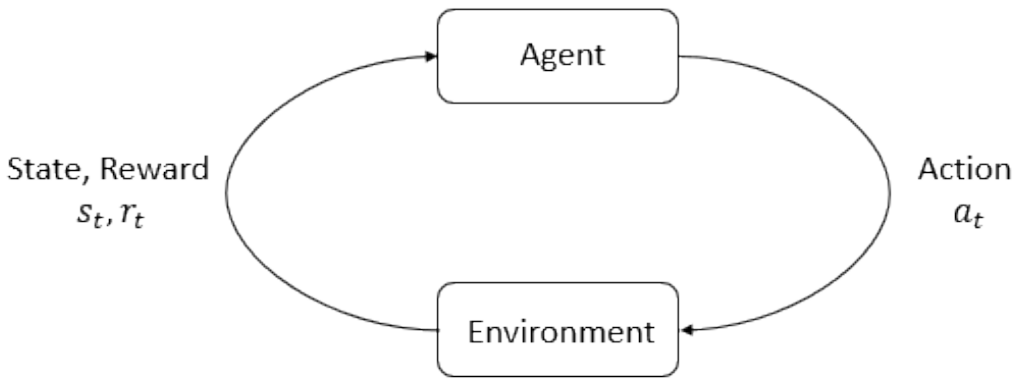


Figure 2.1: Spinning up - Reinforcement Learning Schema

One possible example is a popular learning environment called Pole Balancing, sometimes abbreviated as Cart-Pole. Introduced in (Barto et al., 1983), "this environment has a cart to which a rigid pole is hinged. The cart is free to move within the bounds of a one-dimensional track. The pole is free to move only in the vertical plane of the cart and track" The goal of the agent is to learn to move in such a way that the pole is balanced upright.

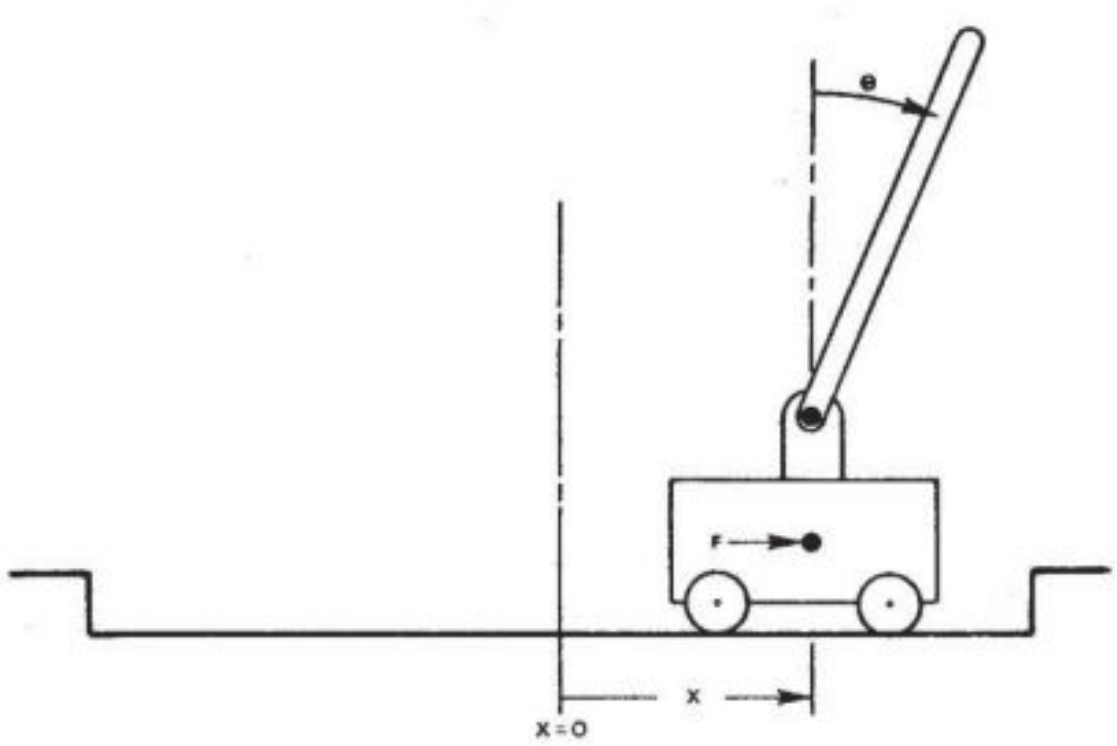


Figure 2.2: Cart-Pole problem (Barto et al., 1983)

Let us use this problem to better describe reinforcement learning

- **Agent** - In this problem, an agent decides how to move the cart left and right.
- **List of Actions** - In this problem Agent has only two possible actions
  - Move to the left
  - Move to the right
- **Environment** - Description of a state of the environment and task to be achieved in that environment - Balancing the pole upright on the cart.
- **State** - Defines the current state of the environment to the agent.
  - Cart Position - At what place in the
  - Cart Velocity
  - Pole Angle
  - Pole Angular Velocity
- **Reward** - How many steps can an agent keep the pole at an acceptable upright angle.

By recording the states that follow each action, the agent can determine which actions lead to which states. This enables the agent to learn how to maximise the rewards it receives.

2.2 Elements of Reinforcement Learning

After explaining what is the basis of Reinforcement Learning. It is time to identify four main subelements of the Reinforcement Learning system according to (R. S. Sutton et al., 2018). These elements are a *policy*, a *reward signal*, a *value function* and optionally, a model of an environment.
Policy defines how an agent chooses its actions. It is what defines what action should be done in what state. For this reason, a function representing policy is often defined as a state-action pair function.

Reward signal defines how well an agent achieves the current problem's goal. On each step, an agent receives a reward for its last action. Through maximising this reward agent learns what are good and bad events for the agent and learns how to behave in an immediate sense.

Value function specifies what is good in the long run. "the *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state"(R. S. Sutton et al., 2018). Rewards are given by an environment immediately. Values, on the other hand, must be estimated and re-estimated from observations achieved over the lifetime of an agent.

Model exists only in part of agents. It is an inference of how an environment behaves and will behave. Knowing this model allows any agent to plan for the future. This ability is very powerful, and the existence of a model in an agent is the most basic categorisation of agents.

2.3 Types of Reinforcement Learning Agents

As noted, the main branching point for an RL agent is whether it starts with given or learned rules about how an environment assigns rewards to state transitions. Those agents that do have access to these rules are called model-based agents, and these rules allow such agents to predict multiple steps ahead, which results in better sample efficiency. However, agents rarely have access to the model from the beginning, and most model-based agents have to learn the world model from their own experience, which brings bias into a model. This bias may cause such a model to not work nearly as well in real environment compared to training. Other than agents with access to an environmental model, there are also so called model-free RL agents. "*While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune.*"(Achiam, 2018)
See 2.3 for a simple schema of categories of popular agents.

2.3.1 Model-Based Agents

As already stated, model-based agents have a model of state value function that can be either given right away or learned during early episodes.
A famous example of an agent with an already given model is AlphaZero (Silver et al., 2017), an algorithm created to play chess and managed to achieve superhuman levels of play
Then there are most of the other agents that have to learn the environmental model by themselves. Among some of the better known are these agents:

- World Models (Ha et al., 2018)
  - Focuses on the possibility of training model-free agents only in simulated latent space worlds created by an environmental model.
- Model-Based Value Estimation - MBVE (Feinberg et al., 2018)
  - Like World Models, MBVE proposes using an environmental model to train model-free agent
  - Unlike World Models, MBVE wants to augment real experiences with fictitious ones instead of only using fictitious ones like World Models
- Imagination-Augmented Agents - I2A (Weber et al., 2017)
  - Complete plans gained from an environmental model are given to model-free agent as subroutine - side information.
  - Policy can learn to choose when and if to use plan - makes model bias less of a problem
- Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning also known as MBMF (Nagabandi et al., 2017)
  - Each time the agent observes the environment, it computes every step of an optimal plan, with respect to the environmental model
  - Agent executes the first action of the plan and starts computing a new plan.

2.3.2 Model-free Agents

While model-based agents start with or create an environmental model of how the world itself works to either plan far ahead or to augment data for training. Model-free agents focus on finding the most optimised ways to behave by trial and error. This is achieved either by optimising policies directly, deriving them utilising Q-learning, or using a combination of these methods.

The first method, optimising policies directly, is called Policy Optimisation. Policy optimisation agents explicitly define their policy as $\pi_\theta(a|s)$. This equation consists of action a, state s, policy π and parameter θ. "This policy is optimised by changing the parameters θ, either by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly by maximising local approximations of $J(\pi_\theta)$"(Achiam, 2018) Thanks to the ability of directly optimising what is needed, Policy Optimisation Methods tend to be way more stable and reliable. However, these methods have their negatives.

In Policy Optimisation, most agents utilise on-policy optimisations. This means that during each policy update, only data from the last policy iteration can be used. This leads to an inability to recycle data from old iterations of policies, which translates to low sample efficiency compared to off-policy algorithms like Q-learning. In simpler terms: **Policy Optimisation algorithms try to find the best policy (state-action pair) that maximises the expected cumulative reward over time.**

The second method is called Q-Learning. These methods work by trying to approximate optimal action-value function $Q^*(s, a)$ through approximator $Q_\theta(s, a)$. An objective function based on Bellman equation is typically used. These equations are based on the simple idea of"*The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.*"(Achiam, 2018). This optimisation is almost always performed off-policy. This means that contrary to on-policy optimisations where only the last iteration of data can be used, any data collected during training can be used regardless of the agent's decision-making process. The actions taken by the Q-Learning agent are then given by $a(s) = \arg\max_a Q_\theta(s, a)$. To once again try to explain this simpler: **Q-Learning agents try to find the best value function (action-value pair) to find which action has the highest expected cumulative reward in each state.**

The combination of these methods is very often categorised as Policy Optimisation agents because the one that *acts* is policy. This policy is then *criticised* through values brought forward by value functions. You could therefore say that these methods could be called Actor-Critic methods. "*The term "Actor-Critic" is best thought of as a framework or a class of algorithms satisfying the criteria that there exists parameterised actors and critics. The Actor is the policy $\pi_\theta(a \mid s)$ with parameters θ, which conducts actions in an environment. The Critic computes value functions to help assist the actor in learning. These are usually the state value, state-action value, or advantage value, denoted as V(s), Q(s,a), and A(s,a), respectively.*" (Seita, 2018). As the citation shows, state-action value (Q-learning) can be used in actor-critic methods. But other critics exist too.

Some of the popular agents, along with their basic premise, is described on the next page:

- Q-Learning Agents
  - Deep Q-Network - DQN (Mnih; Kavukcuoglu, et al., 2013)
    - Q-Value function of this agent is represented by Deep neural network
  - Double DQN - DDQN (Hasselt et al., 2015)
    - This agent focuses on the overestimation problem of DQN by dividing the selection of actions and evaluation of action-values between two separate networks
  - Categorical DQN - C51 (Bellemare et al., 2017)
    - Value of state-action (policy) pair is not estimated as a single value, but an approximation of a categorical distribution over a set of discrete values
  - Quantile Regression DQN - QR-DQN (Richter et al., 2019)
    - Quantile regression is used in this agent instead of a neural network to estimate Q-Value function
  - Hindsight Experience Replay - HER (Andrychowicz et al., 2017)
    - HER focuses on improving the learning of agents in sparse reward environments.
    - A modified version of the experience replay mechanism is utilised
      - This allows agents to learn from experiences that were not directly rewarded but gained that reward later on.
- Policy Optimisation Agents
  - Vanilla Policy Gradient - VPG (R. Sutton et al., 2000)
    - Improved implementation of basic gradient algorithm - One of the most basic Actor-Critic algorithms beyond the tabular case.
    - Chosen for implementation and further described in 2.5
  - Advantage Actor-Critic
    - Actor-critic method that executes a set of environments in parallel to increase the diversity of training data
    - Two different implementations
      - Asynchronous - A3C (Mnih; Badia, et al., 2016)
      - Synchronous - A2C (Wu et al., 2017)
  - Trust Region Policy Optimisation - TRPO (Schulman; Levine, et al., 2015)
    - A variation upon basic Policy gradient that allows for greater learning steps by implementing a KL-Divergence constraint
    - Implementation of constraint very computationally complex
  - Proximal Policy Optimisation (Schulman; Wolski, et al., 2017)
    - Different implementations of the original idea behind TRPO
      - Penalty - Making the original constraint defined in TRPO a penalisation instead of a hard constraint
      - Clipping - Two different policies are run one after another. If the new policy gets too far from the old one, it gets clipped. This discourages large policy changes.

        * This algorithm was chosen for implementation and will be further described in 2.6
- Hybrid Agents - Combines ideas of Policy Optimisation and Q-Value
  - Deep Deterministic Policy Gradient - DDPG (Lillicrap et al., 2015)
    * Learns both Deterministic state-action policy and Q-Value function to evaluate policy
    * Uses a combination of experience replay and target networks to stabilise learning
  - Twin Delayed DDPG - TD3 (Fujimoto et al., 2018)
    * Addresses some stability issues that can arise during training
    * Utilises two Q-Value functions and a more conservative update rule for policy
    * Addresses over-estimation problem by target policy smoothing
  - Soft Actor-Critic - SAC (Haarnoja et al., 2018)
    * Optimisation of trade-off between expected reward and entropy of policy - encourages exploration
    * Utilisation of temperature parameter to control degree of stochasticity - adapts to changing environments

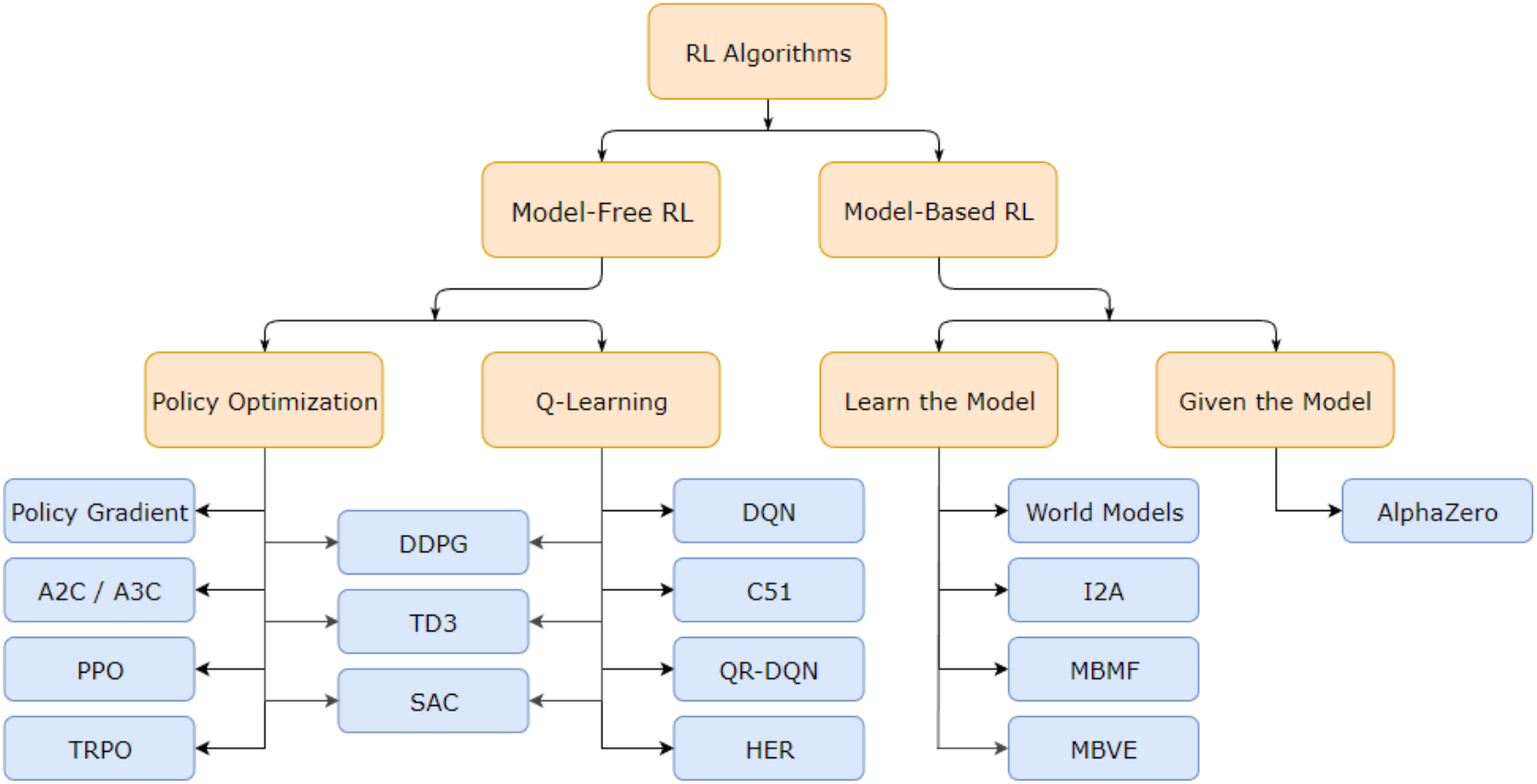


Figure 2.3: Spinning Up - Reinforcement Learning agents

2.4 Policy Gradient Agents

Policy Gradient Agents are a type of Policy Optimisation agents that update their policy by using the gradient of expected cumulative reward with respect to the policy parameters. Gradients help to locate a local maximum or minimum of a function."*Computing the gradient is tricky because it depends on both the action selection and the stationary distribution of states following the target selection behavior. Given that the environment is generally unknown, it is difficult to estimate the effect on the state distribution by a policy update.*", Instead a derivative defined by *Policy Gradient Theorem* (R. S. Sutton et al., 2018) is used. This derivative allows for majorly simplified computations. After estimating gradient, the next step is to use it to update the parameters of the agent's policy. A standard Gradient Ascent or something more complex like ADAM (Adaptive moment estimation) can be used for this. Standard Gradient Ascent is based on a simple iterative change of parameters in the same direction as an estimated gradient of the loss function to maximise the expected return. ADAM, originally defined by (Kingma et al., 2014) is currently a very popular stochastic gradient descent optimisation algorithm used for RL. Unlike standard Gradient Ascent that just moves in the same direction as the gradient, *ADAM* estimates the first and second moment of the gradient by using moving averages of gradient and squared gradient. Using both gradient and squared gradient allows for correcting bias in the gradient estimate and adjusting the scale of the update based on the variance of the gradient. Thanks to this ADAM is more robust and efficient than standard stochastic Gradient Ascent.
For this thesis, Vanilla Policy Gradient (VPG) and Proximal Policy Optimisation (PPO) were chosen. PPO is one of the more popular choices in Reinforcement Learning and improvement of Vanilla Policy Gradient. I have not been able to find the original use of the term Vanilla Policy Gradient, but for this thesis, the term will be used in the context of its utilisation in (Achiam, 2018). In this work, VPG is a variation of an extremely basic REINFORCE agent introduced in (RJ, 1992). By choosing VPG and PPO, we have both basic and more advanced on-policy agents optimising their policies through Policy Gradient methods to compare.

2.5 Vanilla Policy Gradient

Vanilla Policy Gradient was chosen as an example of one of the more basic implementations of Policy Optimisation agents. This work will utilise the implementation defined in (Achiam, 2018). As already stated, Vanilla Policy Gradient is a more advanced and robust variant of REINFORCE agent. This agent got improved with ideas taken from (R. Sutton et al., 2000). The definition this implementation is based on comes from (Schulman, 2016), which brings forward a comprehensive and lucid introduction to the theory of policy gradient algorithms along with pseudocode. Spinning Up implementation of the agent is further enhanced by utilising Generalised Advantage Estimation defined in (Schulman; Moritz, et al., 2015) to better compute the required policy gradient.

Among the most notable differences is how REINFORCE and VPG work with gradient ascent.

While REINFORCE performs gradient ascent once for each action taken for each episode, VPG performs gradient ascent once over multiple estimates. VPG also utilises a baseline such as a value function. The baseline is subtracted from the return in the Policy Gradient Theorem, which reduces the variance of the gradient estimate and helps the algorithm converge faster and more efficiently.

To further explain the principles behind Vanilla Policy Gradient, a pseudocode defined in (Achiam, 2018) will be first introduced in its entirety before explained step by step.

2.5.1 Vanilla Policy Gradient - Pseudocode

Algorithm 1 - Vanilla Policy Gradient Algorithm

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

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta (a_t \mid s_t) |_{\theta_k} \hat{A}_t.$$

7. Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k$$

   or via another gradient ascent algorithm like ADAM.
8. 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.
9. **end for**

Before explaining the first step, we must first explain how Policy and Value functions work together in Vanilla Policy Gradient. For explaining policy and value functions, theory from (R. S. Sutton et al., 2018) is used and simplified for easier understanding.
Vanilla Policy Gradient utilises two neural networks. First is the Policy network - The policy network defines probabilities of every possible action taken from the current state based on its current state bringing the highest reward. This is the part of the agent responsible for acting - it is the Actor in the Actor-Critic definition of the VPG agent, and by updating this network, the agent works towards maximising the rewards received by actions done by this network.

The second network is called the Value network - A network specifying state-value function V_{ϕ_k} that estimates the expected return to be received, starting from that state. This network behaves as the Critic in the Actor-Critic definition, telling the Actor how well it did. This network is trained to minimise the mean squared error between the estimated value to be received in a state and actual value received.
Together these two networks work to improve the performance of the agent. The Policy network selects actions with the highest probability of maximising reward based on the current state, and the Value network informs about the quality of selected action to allow further improvement of the agent's Policy network. By optimising both Policy and Value networks, the agent can achieve an optimal policy that maximises the expected total reward over time. Before an agent can begin training these networks, they require initial values. This is done in step one of the pseudocode.

The second pseudocode step starts the agent's main loop. Each run of this loop defines a single training timestep of the agent.

The third step handles the collection of information. By allowing the agent to run its current Policy in an environment for a specific amount of steps or until it dies a set of trajectories are collected to be used in further training.

In the fourth step *Rewards to go* are calculated. "*Agents should really only reinforce actions on the basis of their consequences. Rewards obtained before taking an action have no bearing on how good that action was: only rewards that come after.*" (Achiam, 2018). Implementation of this principle is called "*Rewards to go*" and utilised in this step to calculate expected return at each timestep.

The fifth step calculates *advantage estimates* - Unlike value estimates that calculate the expected sum of rewards given a specific state, advantage estimates guess how much better or worse an action is compared to an average action. VPG calculates the advantage function by subtracting the state value from the action value at each timestep. By utilising advantage estimates, this agent can focus on actions that are better than average and avoid those worse than average. Usually, this is achieved through GAE - a *Generalised Advantage Estimate*. GAE combines multiple estimates and is very popular for Reinforcement Learning. Adding a bias term to estimate a scaled version of the previous estimate reduces the high variance usually returned in estimating the advantage function. As this work focuses on agents and not their underlying techniques for further information about this principle, please consult the original article (Schulman; Moritz, et al., 2015).

The sixth step focuses on estimating the actual Policy Gradient. Gradients are useful as they allow us to find the steepest increase of returned rewards for a given input. The majority of logic behind Gradients can be found in Policy Gradient Theorem, which is also explained in

(R. S. Sutton et al., 2018). VPG specifically uses one of the derivatives achieved from Policy Gradient Theorem that (Achiam, 2018) calls "*Expected Grad-Log-Prob Lemma*" or EGLP in short. This Derivation of Policy Gradient Theorem allows us to very efficiently compute the expected gradient of the log-probability of an action, with respect to the Policy, and can be found in the equation as $\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta (a_t \mid s_t)$. This part of the equation is the part that computes the gradient of the log-prob of a trajectory. To EGLP, a baseline is added in the form of Advantage function $\hat{A}_t$ to reduce the variance of gradient and allow the Policy to converge faster and more efficiently. So far, this part of the equation has created an expectation of gradient. It can then be estimated with a sample mean represented in this equation as a mean of a set of trajectories $\mathcal{D}_k$. The derivation process to achieve this equation can be found in chapter nine of (Achiam, 2018).

The next step, the seventh, applies the estimated policy gradient to our policy network, either through standard gradient ascent or ADAM. I already talked about the theory behind this part at the beginning of 2.4.

The final technical step, the eighth one, updates the existing value function by utilising regression on mean squared error. This equation calculates this error by subtracting the actual reward return from the expected return defined by the value function. This error is then mean-squared across multiple samples of trajectories. This is then applied to the existing value function, similarly to step seven. However, unlike Policy where we want to maximise probability, in the Value function we want to minimise error, so a gradient descent is used instead. Once again, either standard gradient descent or something more complex like ADAM.

The final Actual step in pseudocode purely closes the main loop.

In summary, Vanilla Policy Gradient utilises a combination of two neural networks, Actor and a Critic. Based on his current knowledge, the Actor chooses actions in a set of steps with the highest probability of maximising reward. Critic criticises his actions while taking notice of the difference between his expected reward from actions taken and the actual reward gained from these actions. After every set of steps, both of them improve themselves. The Actor uses the Policy Gradient Ascent with an Advantage estimate for better stability and efficiency to maximise the probabilities of an action achieving the highest reward and Critic uses the Policy Gradient Descent to minimise his eror in predicting the value gained from an action.

2.6 Proximal Policy Optimisation

Before we can start explaining the theory behind *Proximal Policy Optimisation*, we must look into the source of the root ideas that *PPO* implements through different means. This agent is called *Trust Region Policy Optimisation*, first proposed in (Schulman; Levine, et al., 2015).

2.6.1 Trust Region Policy Optimisation

With *Trust Region Policy Optimisation*, a way is proposed to improve performance by allowing larger steps in policy optimisation. Originally this was very problematic as a single bad step could have collapsed the policy performance. This problem was fixed by proposing a special constraint that defines how close the new and old policies can be. KL-divergence, also known as relative entropy, is introduced to enable this constraint. The full name of this constraint is *Kullback-Leibler Divergence*, and it is a type of statistical distance that measures the difference between two probability distributions. More specifically, how much information is lost when one is used to approximate another. This allows for measuring the differences between probability distributions, limiting steps that differ too much from the original policy.

However, the theory of *TRPO* requires complex calculations that are quickened by approximations. Through this, however, a problem in optimisation of the approximate is raised, and many steps have to be taken to solve and sidestep this problem. This leads to the agent being fairly slow and processing intensive. Yet, the original idea of *TRPO* had a lot of merits, and from its metaphorical ashes, *PPO* was born.

2.6.2 PPO - basic theory

First mentioned in identically named article by (Schulman; Wolski, et al., 2017), *Proximal Policy Optimisation* utilises simple tricks to fix the same problems as *Trust Region Policy Optimisation* - problems that do not allow taking larger steps in policy updating. Compared to *TRPO*, this agent offers a similar ability for larger steps at way lower computational costs by avoiding a complex secondary function present in *TRPO*.

Two different variants of Proximal Policy Optimisation exist: *Clipped Surrogate Objective*, further referred to as *PPO-CLIP* and *Adaptive KL Penalty Coefficient*, further referred to as *PPO-Penalty*.

2.6.3 PPO-Penalty

In this variant, a KL-constrained update is approximated through several iterations. But instead of making this constraint a hard one, blocking any divergent steps, the KL-divergence is penalised in objective function instead.
An adaptive KL penalty coefficient is used and adjusted during training based on the KL-divergence specifying the distance between new and old policy. If KL-Divergence increases by more than the pre-specified threshold the penalty is made stronger by increasing the coefficient. Similarly, if KL-Divergence is decreased too much, the penalty coefficient is decreased to lower the strength of the penalty.

It is a powerful technique. However, along with this technique, another variation appeared: *Clipped surrogate objective*. And as is said in the original article: "In our experiments, we found that the KL penalty performed worse than the clipped surrogate objective" (Schulman; Wolski, et al., 2017)

As such focus of this work will be on PPO-Clip, and PPO-Penalty will not be used or explained more in-depth in this work.

2.6.4 PPO-Clip

The better and more commonly used alternative to PPO-Penalty is PPO-Clip. "*PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialised clipping in the objective function to remove incentives for the new policy to get far from the old policy.*" (Achiam, 2018)

As explained in article *RL — Proximal Policy Optimisation (PPO) Explained* (Hui, 2018) This is achieved through maintaining two policy networks. One is current, which you want to refine, and another, which you last used to collect samples. Thanks to this, it is possible to evaluate new policies with samples collected from older policy.

To prevent inaccuracies, it's necessary to synchronise the old policy every few iterations. This is done by comparing the new policy to the old policy and calculating the difference. If the new policy is significantly different from the old policy (outside of the range of $(1-\epsilon, 1+\epsilon)$), a new objective function is created to limit large policy changes and ensure more accuracy.

This thesis uses the Spinning Up implementation of PPO-Clip that utilises a simplified derivation of the original expression is used. Original PPO-CLIP defined by (Schulman; Wolski, et al., 2017) is calculated as follows:

PPO-clip updates policies via

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L\left(s, a, \theta_k, \theta\right) \right],$$

(Achiam, 2018)

Typically taking multiple steps of (usually minibatch) Stochastic Gradient Descent to maximise the objective. In this equation, symbol E stands for the expected value operator. It is used to denote the expected value of the surrogate objective function L, with respect to the current policy distribution pi_theta_k, which is defined by the current set of policy parameters $theta_k$. The expectation is taken over all possible state-action pairs (s, a) that can be sampled from the policy distribution pi_theta_k. The surrogate objective function is used as a proxy for the expected reward as this value is difficult to directly optimise in Policy Gradient Methods.

Here L is given by

$$L_{\theta_k}^{CLIP}(\theta) \doteq \mathop{\mathrm{E}}_{s,a\sim\theta_k}\left[\min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}A^{\theta_k}(s,a), \operatorname{clip}\left(\frac{\pi_\theta(a \mid s)}{\pi_{\theta_k}(a \mid s)}, 1-\epsilon, 1+\epsilon\right)A^{\theta_k}(s,a)\right)\right]$$

In this equation, the Expectation E is taken over states and actions encountered by the previous policy. $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ checks how different is the new policy π_θ and the old policy π_{θ_k} and how much more likely would the new policy select action a in state s over the old policy. $A^{\theta_k}(s,a)$ defines the Advantage function, which estimates how much better action a is than the average action in state s, according to the previous policy π_{θ_k}. And ϵ is a (small) hyperparameter that roughly says how far away the new policy can go from the old.

All of these parts of the clipped surrogate objective L come together in expectation over smaller from two terms. The Unclipped Surrogate Objective that measure how much the new policy improves over the old one scaled by the Advantage, and the clipped version of the first term that limits the size of an update to be no more than $(1-\epsilon, 1+\epsilon)$. This ensures that any policy update is not too large.

This original expression got derived by (Achiam, 2018) into the following equation:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right),$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

While the main part of the equation is quite similar to the original, the clipped part of the term was replaced by a clipping function g that modifies the Advantage estimate A. More specifically, if A is positive, g scales A up by a factor of $1+\epsilon$, and if A is negative, then it is scaled by a factor of $1-\epsilon$. Through this, a clipped advantage estimate $g(\epsilon,A$ is used instead of the original Advantage estimate A in the update rule for the policy parameters.

After explaining the theory behind PPO-Clip, pseudocode comes next:

2.6.5 Proximal Policy Optimisation - Clipped Surrogate Objective - Pseudocode

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

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

   typically via stochastic gradient ascent with ADAM.
7. 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.
8. end for

As you can probably see, pseudocode for PPO is almost identical to the one for VPG except for step six being merged with step 7. The original estimation of Policy gradient got replaced by updating the policy through the mean of the PPO-Clip objective over a set of samples of trajectories, through some gradient ascent or more complex gradient functions like ADAM. As the only new equation already got explained in the basic theory behind the PPO agent, it will not be repeated here.

With this, we have introduced the theory behind Policy Optimisation agents, their subcategory Policy Gradient Agents and the specific agents chosen for implementation. The next chapter will give a deeper dive into the theory behind AIQ.

3. Upgrading AIQ test to Python 3

As many years have passed since AIQ was originally implemented, technology has changed. This work was initially created using Python 2, which is now outdated. Python 3 was released in 2008 to address design errors present in Python 2 and included many changes that were not backwards compatible. As of 2020, development support for Python 2 has been discontinued. Even Pip, a package manager that assisted in acquiring, installing and maintaining packages required for your projects, is no longer available, and any possible packages have to be painstakingly managed manually. Due to these reasons, it was necessary to port this algorithm from Python 2 to Python 3.

This chapter will begin with a description of the original prototypical implementation of AIQ using Python 2 in section 3.1 followed by section 3.2 defining what steps were taken to enable AIQ to work on Python 3.8. Section 3.3 will introduce improvements to the AIQ prototype other than updating to newer versions of Python. Finally, section 3.4 will describe basic tests taken to ensure that both old and new versions of AIQ have comparable results.

3.1 Original Implementation

The implementation (Legg; Veness, 2011a) of the Algorithmic Intelligence Quotient consists of four main parts: agent package, reference machine package, the algorithm for computing AIQ from logs and the main AIQ algorithm. Each of these parts will be explained in the following subsections.

3.1.1 The Agents Package

The “agents” package contains all implementations and wrappers of agents implemented for requirements of the original conference paper (Legg; Veness, 2011b).

All of these agents inherit from an abstract class Agent.py which defines the required attributes and methods for all agents, along with a couple of helpful methods that agents can use. From the required parts, the most important ones are information about the observation and action space of the reference machine and methods for resetting the agent and acting in an environment based on observation and reward of the last action.

In the original implementation, the agents implemented were the following:

- Manual - Simple interface enabling human users to pass actions to AIQ like an agent.
- Freq - Simple agent alternating between a random action and an action with the best mean reward.
- Random - Trivial agent passing random actions to AIQ.

- Q_l - a table based Q-learning with eligibility traces as written in (R. Sutton et al., 2000), can also run Q_0 by defining Lambda=0 as originally introduced in (Watkins, 1989).
- HLQ_l - A similar agent to Q_l with the addition of automatic optimisation of its learning rate created by Legg and Hutter (Hutter; Legg, 2008).
- MC-AIXI - A wrapper for a more advanced agent utilising an approximation of Hutters AIXI defined in (Veness et al., 2010).

3.1.2 The Reference Machine Package

Next package, *"refmachines"* contains classes for interaction with defined reference machine. Currently, the only reference machine defined is BF in class "BF.py". This class has two main uses:

- Processing of given BF programs and passing information about the environment to the main algorithm. Specifically through computing the reward and observation (output) tape from instructions given by the program.
- Generation of a random program. This includes sanitisation of some pointless code like empty loops and reward-insensitive code like "heaven"(always full reward) and "hell" (always no reward) in samples. This random program is later classified into various strata in another class, "BF_Sampler.py" and saved as a sample to be utilised.

The BF code interpreter and all future interpreters have to inherit from a parent class "ReferenceMachine.py" to allow interaction with the main AIQ code described in subsection 3.1.3. This package also contains the folder "samples" where AIQ code expects to find generated samples for testing.

BF.py can also negate all rewards if required through a parameter. This allows for the application of antithetic variance reduction defined in 1.4.1.

3.1.3 The AIQ Algorithm

The main algorithm "AIQ.py" works as a bridge between a reference machine, a user and an agent. Through this algorithm, the defined sample is loaded and given to an estimator. While both a simple and a stratified estimator are implemented, a simple estimator doesn't support logging, which is required for computing the AIQ. It is useful for debugging, but a stratified estimator is the one for actual practical use.

The stratified estimator takes loaded program samples and differentiates them according to different strata to ensure each stratum is present in the final set of programs. After preparing the set of programs, data is passed to various threads of computational devices where agents are asynchronously engaged with the same program twice. Once with positive and once with negative rewards to ensure that random decisions of agents result in a cumulative AIQ of 0.

Before being passed to the agent, a reference machine interprets the program, and an agent only receives information about new observations and rewards from the last step. If logging is enabled, the results are logged into a file along with time, stratum and rewards of a positive and negative run.

3.1.4 The *Compute_from_Log* Algorithm

The last algorithm, "ComputeFromLog.py" loads and processes referenced log file and, through data contained within, calculates the final AIQ score.

3.2 Python 3 Update

Thanks to a relatively small volume of code present in the AIQ project, it was possible to implement the update to Python 3 through a manual check of the entire code. By taking inspiration from (Brett Cannon, 2013), I have created a list of changes causing the most problems:

- Print is no longer a Statement but a function.
- Different commands used in command line manipulation:
  - Instead of raw_input, input is used in Python 3.
- Division operator / had its behaviour changed when working with Integers:
  - In Python 2, operator / worked as Integer division when variables were Integers and returned a value rounded to the nearest integer. This is no longer the case in Python 3, where this operation now returns a float approximation of the result. To fix this, the rounding division operator // was used in places where integer division is utilised.
- Changes when working with Strings:
  - New String formatting, many functions originally in separate packages implemented as a base part of Python.
- Many functions of the Scipy package transferred into its successor package Numpy.

I have successfully fixed these problems and created a new working version of AIQ.

3.3 Additional Improvements

I began working on the AIQ test but encountered some additional issues that required attention. One of those problems was a difference between handling Global variables in multithreading across various Operation Systems. On Windows, for example, global variables have not been passed to new threads, which resulted in the inability to log or read any defined options. To fix this problem, I have switched from using global variables to creating a dictio-

nary (associative array) containing all necessary information originally saved globally. This dictionary was then passed as a parameter of classes running across different threads.

Next, I added a debugging option that outputs every unique observation and action symbol passed to an agent during an episode.

I have also encountered agents failing in specific situations during testing, causing the whole AIQ test to halt and "crash". A parameter defining whether an agent is still alive has been added to deal with failing agents. If an exception is raised, an agent is defined as no longer alive and logged in the log for further analysis. After this, the agent is reset to begin learning from the beginning. There are many ways to handle these problems differently: Penalising an agent to always return "-100" after dying, making an agent neutral after dying by always returning a reward "0", but doing that could affect AIQ score too much.

3.4 Evaluating the New Version

After implementing fixes for these problems, tests have been made to ensure no major result difference due to incorrect code.

3.4.1 Basic Functionality Test

First, a basic "function" test has been done to check if all program functions work as intended. Through running commands that checked:

- AIQ test and all of its parameters.
- ComputeFromLog.py both basic and full.
- Generation of BF samples and all its parameters

I have confirmed that all of the functionalities of the original AIQ test in Python 3 have been kept. Additionally, I have run the AIQ test on the original sample for all original agents at thousand episode length five times at same configurations as provided by (Legg; Veness, 2011a) file "*Conf Paper Settings.ods*".

While these extremely basic tests returned similar distribution of results to the original conference papers, the newly acquired results were about 1 AIQ score higher on average for all agents. As there were some differences, it was required to perform a more advanced test to check that the update was performed correctly.

3.4.2 Statistical Test of Difference

With the assistance of (Vadinský, Unreleased), I have gained access to data of original agents on the same samples in both the original AIQ test implementation in Python 2 and my new implementation of the AIQ test in Python 3. With these results, it is possible to perform a statistical test to find whether there is a significant difference between data from the original implementation of agents compared to my implementation. The Paired T-test is commonly used to compare two groups before and after an intervention, making it ideal for our purposes.

Hypothesis

The first thing to define while performing a statistical analysis is the hypothesis. As we need to check whether the AIQ values of agents in Python 3 are different compared to the AIQ values of agents in Python 2 we can define the following hypothesis:

- H0: There is no significant difference between the final AIQ score at 100 000 steps of tested agents at the Python 2 version of the AIQ test and the Python 3 version of the AIQ test
- H1: A significant difference exists between the final AIQ score at 100 000 steps of tested agents in the Python 2 version of the AIQ test and the Python 3 version of the AIQ test

Data

For comparing the agents we utilise all configurations of the original agents of HLQ_l, Q_l, Q_0, and Freq provided by (Vadinský, Unreleased). We are mainly interested in the AIQ score at 100 000 steps. The data we used can be found in 3.1. The columns "Pyt_2" and "Pyt_3" contain values of various agents configurations at 100 000 steps, column "Difference" contains the calculated difference between acquired AIQ scores and "Pyt_2_MOE" contain the margin of error calculated by the original version of the AIQ test

Analysis

Already from the data, it can be seen that the difference in AIQ score between the different versions is smaller than provided margin of error, which already supports our null hypothesis of there being no significant difference between the different versions of the AIQ test. But for additional support, a paired T-test will be performed.

Agents	**Pyt_2**	**Pyt_3**	**Difference**	**Pyt_2_MOE**
(Freq,0.03)	56,4	56,6	-0,2	0.5
(Freq,0.05)	57	56,5	0,5	0.5
(Freq,0.07)	57	56,8	0,2	0.5
(Freq,0.09)	56,5	56	0,5	0.4
(Freq,0.11)	55,5	55,4	0,1	0.4
(H_l,0.0,0.0,0.95,0.04,0.7)	61,8	61,6	0,2	0.4
(H_l,0.0,0.0,0.99,0.02,0.7)	64,8	64,8	0	0.4
(H_l,0.0,0.0,0.99,0.04,0.6)	63,2	63,3	-0,1	0.4
(H_l,0.0,0.0,0.995,0.005,0.9)	65,3	65,1	0,2	0.5
(H_l,0.0,0.0,0.995,0.01,0.8)	65,6	65,6	0	0.5
(Q_l,0.0,0.0,0.5,0.005,0.95)	62,9	63	-0,1	0.5
(Q_l,0.0,0.0,0.5,0.01,0.9)	62,7	62,3	0,4	0.5
(Q_l,0.0,0.0,0.5,0.02,0.8)	60,3	60,3	0	0.4
(Q_l,0.0,0.0,0.5,0.03,0.7)	57,8	58	-0,2	0.4
(Q_l,0.0,0.0,0.5,0.04,0.6)	55,5	55,4	0,1	0.4
(Q_l,0.0,0.5,0.5,0.005,0.95)	63,3	63,4	-0,1	0.5
(Q_l,0.0,0.5,0.5,0.01,0.9)	63,3	63,2	0,1	0.4
(Q_l,0.0,0.5,0.5,0.02,0.8)	62	61,7	0,3	0.4
(Q_l,0.0,0.5,0.5,0.03,0.6)	59,1	59,2	-0,1	0.4
(Q_l,0.0,0.5,0.5,0.04,0.6)	58	58,1	-0,1	0.4

Table 3.1: AIQ score at 100 000 steps across different Python versions

To utilise this test, we need to check for assumptions of a paired T-test:

- Independence
- Normality
- Equal Variance

Independence: Is fulfilled as there is no influence between acquired values.

Normality: Requires a normal distribution of data. To check for this assumption, a Shapiro-Wilk test is performed on acquired AIQ scores of Python 2 and Python 3 versions separately. This test was performed using the Python package "scipy.stats" method "shapiro". This method checks the null hypothesis that the data was drawn from a normal distribution. Thanks to this test, I have acquired the following P-values:

- P-value of Python 2 version: 0,06
- P-value of Python 3 version: 0,08

As both of the calculated P-values of the Shapiro-Wilk test are above significance level $\alpha = 0.05$, the null hypothesis of normality is not rejected.

Equal Variance: Requires the variance of differences between the data pairs to be equal. I have checked this using the Levene test for equality of variances. Using the "levene" Python method from the package "scipy.stats" I have checked the null hypothesis that all input samples are from populations with equal variances. Using this method with the acquired data of both Python 2 and Python 3 versions, I have acquired the following P-Value:

- P-value of Levene's test: 0,99

As the calculated P-value from Levene's test is above significance level $\alpha = 0.05$, the null hypothesis of equal variances is not rejected.

With this, we have checked all the assumptions of a paired T-test.

Paired T-test: Can be tested using the method "ttest_rel" from Python package "scipy.stats". We can use the Levene's test to test our null hypothesis by using the AIQ score of agents from different versions of Python as parameters. This will help us obtain the P-value. The returned P-value from this method is:

- T-statistic of paired T-test: 1,76
- P-value of paired T-test: 0,09

As the calculated P-value of the paired T-test is above significance level $\alpha = 0.05$, the null hypothesis of no significant difference between the results of the final AIQ score at 100 000 steps is not rejected.

Discussion

During our analysis of data acquired from running the original implementation of agents on Python 2 and the new implementation of agents on Python 3 on the same samples, no support has been found for significant differences between data.

The difference between acquired values belongs to the limits defined by the margin of error. Performing a paired T-test also returned a P-value higher than significance level $\alpha = 0{,}05$, which further supports the null hypothesis of there being no significant difference between acquired data.

4. Implementing Policy Optimisation Agents into AIQ Test

After reviving the AIQ test into its new form that is no longer running on a deprecated Python version, it is finally time to begin what was the original purpose of this thesis. Enabling AIQ test to work with chosen policy gradient agents Vanilla Policy Gradient introduced in subsection 2.5 and Proximal Policy Optimisation in subsection 2.6.

Since I have little experience with implementing Reinforcement Learning agents, my objective was to locate and utilise a well-known package to ensure the accuracy of the implementation. Unfortunately, I had to slightly alter this plan as I encountered various problems making this original plan difficult. These problems forced me to heavily modify an existing Python package to allow compatibility with the AIQ test.

This chapter will first focus on introducing packages I have considered as candidates for implementation, along with an introduction of a standardised framework for environments.
I will then present problems that stopped me from using the packages in their unmodified state. The chapter's final two sections will focus on agents' implementations. First a description of the architecture behind the Spinning Up package, then the new architecture introduced to allow for compatibility with AIQ.

4.1 Brief introduction of used RL agent packages and OpenAI Gym

There are two main frameworks upon which Reinforcement Learning agents can be built: Tensorflow (Martín Abadi et al., 2015) and Pytorch (Paszke et al., 2019). As openAI recently moved to Pytorch as its standard framework (OpenAI, 2020) I have decided to also base my code on Pytorch.

After deciding what framework to use, I searched for popular Python packages providing code of RL agents. I have looked into three widely known packages: Spinning Up (Achiam, 2018), Garage (contributors, 2019) and Stable Baselines 3 (Raffin et al., 2021) and later on got inspired by minimalistic implementation from "minimalistic implementation of Vanilla Policy Gradient" by(Barazza, 2019). Finally, this chapter will introduce OpenAIGym (Brockman et al., 2016), a standardised framework for RL environments.

4.1.1 Spinning Up

Spinning up (Achiam, 2018) is widely recommended as an introductory step for programmers interested in Reinforcement learning. Their documentation contains understandable explanations of principles behind many basic Reinforcement Learning principles. Code in this library focuses on simplicity and ease of use, making it accessible to both researchers and practitioners.

Aside from the basic principles of RL, Spinning Up also contains simple implementations and explanations of the theory behind some popular RL agents. The code of these agents was initially based on the currently deprecated Tensorflow 1, but many were modified to be compatible with Pytorch later on.

Unfortunately, this package was not maintained for the last three years, so any problems found during these years were not fixed. During my testing, I ran across one such problem that made any testing of the AIQ environment through the OpenAI Gym environment impossible.

4.1.2 Stable Baselines 3

Stable Baselines 3 (Raffin et al., 2021) is the newest iteration of the RL agent package based on the original baselines (Dhariwal et al., 2017) package released by OpenAI. OpenAI did this to literally create a research"*baseline*" that would allow for easier replication, identification and refining of new ideas. This package was later on forked into new and improved "*stable-baseline*"(Hill et al., 2018) and later rewritten from Tensorflow to Pytorch in Stable Baselines 3 (Raffin et al., 2021).

Stable Baselines 3 is a highly popular and actively maintained package that provides reliable and high-quality implementations of classic and state-of-the-art reinforcement learning algorithms. Stable Baselines 3 contain in-depth documentation and have almost full test coverage, and their results are compared to benchmarks of previous versions.

4.1.3 Garage

Garage (contributors, 2019) is a toolkit initially based on package rllab (Duan et al., 2016). After the end of maintenance for rllab, a team of researchers from several universities created garage as its successor.

The garage is still actively maintained and provides a flexible and modular framework for developing, testing and evaluating RL algorithms. Garage aims to empower researchers and practitioners by offering a range of tools and algorithms to facilitate RL research and development.

4.1.4 Minimalistic Vanilla Policy Gradient

Minimalistic Vanilla Policy Gradient implementation by (Barazza, 2019) Is not a package, merely an extremely minimalistic implementation of a simple policy gradient agent in a single file. I am unsure if I would even call this implementation Vanilla Policy Gradient as it lacks an actual advantage function calculated from the value function and purely uses rewards-to-go for its advantage calculation.

While not a package, the extreme simplicity of this implementation provided me with a key for understanding and unlocking the meaning behind much of the code of the other packages. For this reason alone, I have included a reference to this work as an honourable mention.

4.1.5 OpenAI Gym

As you might remember from chapter 2 Reinforcement learning consists of interactions between agents, provided by packages and toolkits, and environment. I found that all the packages and toolkits utilise a standardised framework and toolkit for RL environments called OpenAI Gym (Brockman et al., 2016).

OpenAI Gym provides an abstraction of interaction with an environment. Thanks to this abstraction, it became possible to focus on developing purely one side of the interaction between an agent and an environment at once. Additionally, any environment created could be used in any RL agent that used the common interface from provided toolkit.

This toolkit also included a growing collection of environments to be used as benchmarks. All of these environments share a common central interface. Thanks to this, there was no need to create an environment for every agent, and researchers could focus purely on developing RL agents. This interface became a standard that was either used or made compatible with the majority of publically available RL agent toolkits.

Unfortunately, the reliance on this framework for all mentioned packages led to a significant problem that made it impossible to utilise the packages for the AIQ test without major alterations. Specifics of this problem will be described in section 4.2

In October 2022 all further development on OpenAI Gym has been moved to a new library called Gymnasium (Foundation, 2022).

4.2 Problems in implementation

While I analysed and experimented with implementing packages mentioned in section 4.1 into the AIQ test, I ran across many problems. One of those forced me to explore multiple dead-end paths of code experimentation before I was forced to acknowledge that I could not

implement these packages unmodified.

I will first introduce the architecture problem that made me work on modifying the package purely for AIQ purposes. Next, I will introduce some other roadblocks I had to overcome during the mentioned package modification.

4.2.1 Interaction with Environment

At the core of every program lies architecture. While some programs are a mess of entangled processes, programmers usually strive towards an effective and clean code structure. Well defined architecture can ease any future work in an algorithm and allow for cooperation between different authors. OpenAI Gym introduced one such architecture for interactions between agents and environments.

Gym is designed to allow agents to simply accept a Gym environment and use it to learn through thousands of epochs and hundreds of episodes at once by itself. Unlike Gym based architecture that handles interactions with agents on hundreds and thousands of interactions at once, AIQ has to simulate every step of the environment after every action of an agent. This architecture of AIQ is entirely inverse to the one Gym is based on and required me to devise a way to fix it. For a better understanding of the architectural differences please refer to image 4.1.

Some of the experiments I have tried contained an additional AIQ refmachine compatible with OpenAI Gym interface or a Custom Dummy environment with a method that would allow the environment to accept information about itself from outside every step. Yet, none of these experiments managed to achieve an imitation of OpenAI Gym environment interface to a level that would allow direct connection with agents imported from packages. I believe that it is possible to rewrite AIQ into OpenAI Gym compatible environment or create a custom Wrapper that implements compatibility with this environment. Unfortunately, to my understanding, such a task would require a complete rewrite of the AIQ implementation, which is outside this thesis's scope. However, It is a fascinating concept that would deserve future research as compatibility with AIQ gym would allow for massively easier implementation of widely used agents for evaluation.

In the end, I was forced to create a manual implementation of VPG and PPO agents with the complete removal of the OpenAI Gym interface. For this, I have chosen the package that focuses on simplicity and ease of use: Spinning Up (Achiam, 2018). The simplicity of the code allowed me to understand and implement agents into AIQ without needing to completely rewrite massive parts of code libraries.

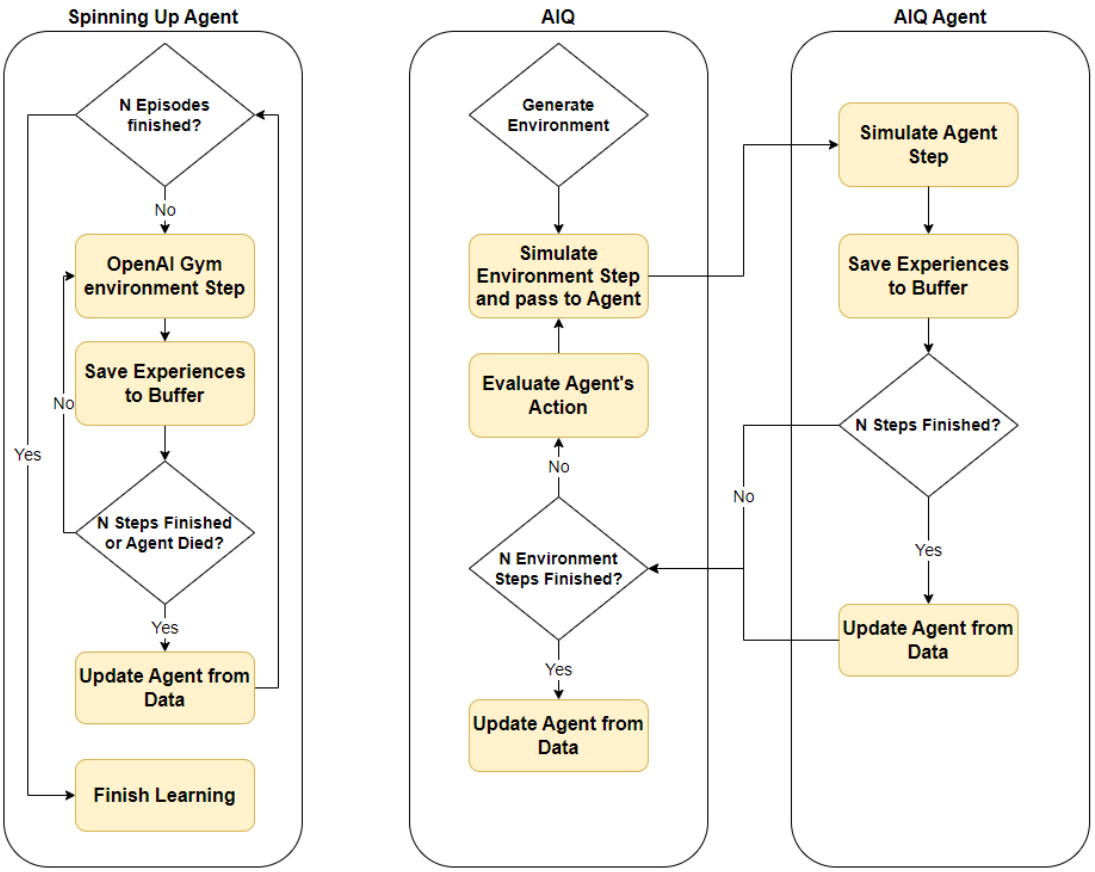


Figure 4.1: Environmental Interaction - OpenAI Gym and AIQ

4.2.2 Representation of Environment

Once I decided on the package to use, it was time to remove the interface of OpenAI Gym from the package completely. During this removal I encountered another problem. The OpenAI Gym interface has its own classes that define how all the data (Observation, Action, Reward) an agent needs is stored. These classes have to support various dimensions (Discrete, Continuous, Box and other), so they are fairly complex. While AIQ only needs support for the Discrete representation of both Action and Observation, I did not know the format an agent, its neural network, and its experience buffer require from its data. Integer? Float? Numpy Array? Pytorch tensor? As there are complex mathematical steps requiring very specific data formats, it took me months of research and experimentation to find an answer.

My attempts were hindered by the unpredictable behavior of the AIQ reference machine that sometimes sent data in different formats. To address this issue, I used one-hot encoding to transform the incoming data into a binary format that could be more easily processed by the neural network. This involved assigning a unique binary value to each possible category within the data, creating a matrix where each row represented an observation and each column corresponded to a category. By representing the data in a more structured and uniform way, one-hot encoding allowed for more accurate analysis and better results. The agent then returned the action as an integer within an array, which, along with the reward passed from outside as an integer and one-hot encoding, was saved in an inner buffer for later use in training and updating the gradient.

4.2.3 Logging

Spinning Up utility packages logged most of the information to a standard output, which is unacceptable for use in AIQ, where such information would clog the standard output of AIQ containing important data. The data passed by the inner working of data nonetheless contain some very interesting information about the process of the agent's learning. This led to altering the Spinning Up logging libraries to move from direct command line output to creating an inner database of agent's learning data. This inner database contained methods allowing this data to be exported to a file once a specific parameter was passed with a call to AIQ.

4.2.4 Agent Failures

In specific situations, an agent can fail. For example, there are some environments where the advantages between policy changes are incredibly close to 0. With machine precision problems, this sometimes leads to inner algorithms dividing by zero and returning NaN. With NaN in place, where a number is expected, Agent rapidly fails and, unless caught, manages to destroy the entire run. This can happen for many reasons: too low learning rate, too high learning rate, too small experience buffer and many others.

However, even if this problem is caught, a question arises. If an agent fails, how should the rest of the run be calculated for the resulting AIQ? Should it always return 0 to define random behaviour? Should the agent be penalised and always return 100?

After multiple consultations, the solution we have come up with is to reset an agent to its initial state in case of failure and save a piece of information that the agent failed in this specific environment. Further research on these faults, possible ways to fix them and how to handle them if they cannot be fixed might be an interesting path for future experimentation.

4.3 Spinning Up Package Architecture

To better understand the code's changes, I will describe the code behind the Pytorch implementation of VPG and PPO agents from Spinning Up package.

Spinning up contains a utility folder in their algorithm file directory that contains helpful methods for logging, multiprocessing, plotting and better search over hyperparameters. All agent code is first separated by their reliant package (Tensorflow and Pytorch) before having a folder for each agent. After being separated, each agent has their own folder consisting of two files. *Core.py* contains everything necessary to initialise Actor (and Critic) neural networks and the other file that is always named after the agent. The file named after an agent consists of an experience buffer class and function that handles the main agent's loop.

4.3.1 Auxiliary Classes and Functions

The focus of this thesis is on VPG and PPO, and as the only difference is in how they handle updating their policies, I will describe the architecture of both of these agents together.

In Spinning Up, main loop of an agent is a function with the same name as an agent. This function accepts the following groups of parameters:

- OpenAI Gym Environment
- Actor Critic and its parameters
- Hyperparameter values of anagent
- Logging and its parameters

To better understand the agent function, I will briefly explain what each of these groups of parameters handles before mentioning the existence of multiprocessing support. Once all of these are sufficiently explained, I will explain the main agent loop.

OpenAI Gym Environment

Open AI Gym provides a massive library of existing environments to be used for research and testing. The agent function requires the following from this environment:

- Shape of action and observation space
  - Defines what an agent sees and how it can act. Through these pieces of information, the size and the dimensions of experience buffer and neural networks can be defined.
- Method for a single step in the environment
  - A method that simulates a single step in an environment with given data when called
- Method for a reset of an environment
  - Method that resets environment to its initial state when called

Actor Critic and its parameters

Actor and Critics are the main parts of the VPG and PPO agents. In Spinning up implementation, their initialisation is defined in the core.py.

This file contains a class MLPActorCritic that, depending on the shape of environment, decides between a Gaussian actor in the case of a Box (two-dimensional) environment and a Categorical actor in the case of a Discrete (one-dimensional) environment. Once the type of environment is decided, neural networks for both Actor and Critic are created along with methods allowing for learning and, in the case of the Actor network, a step method that returns the next action of an agent according to observation data and current state of Actor (policy) network.

Hyperparameter Values

Each agent has the parameters required to define how they should be learning. For more information refer to subsections 2.5 and 2.6

Logging and its Parameters

Spinning Up utilises its own Logging class located in the utility folder, specifically in *logx.py* file. This class is made for logging massive amounts of information about an agent in each epoch. Almost any interesting information, like the loss value of both policy and value functions, the deltas between previous and current loss values and many other pieces of information, get saved before being exported to a standard output and output log file. Another information saved is the entire state of the agent¨'s model. Parameter save_freq defines after how many updates the entire model is saved. the *logx.py* file also contains a method for saving models of both Pytorch and Tensorflow agents.

Multiprocessing

Utilising multiple processors is necessary to improve the efficiency of workstations, clusters, and supercomputers due to the high processing power required by AI. This is where the *Message Passing Interface* (Message Passing Interface Forum, 2021) standard is used. This standard defines how to correctly communicate information between different processors during multiprocessing. Spinning Up utilises a MPI4PY package that allows for utilisation of this

standard and the entire multiprocessing principle in Python. Spinning Up further has multiple additional files that support additional operations across multiple processes, including patches that improve compatibility between MPI and other utilised packages like Pytorch.

Buffer class

Aside from the main loop, the agent file also contains a Buffer class. This class gets initialised with environment sizes and epoch lengths and contains methods for saving and accessing data. This class also contains a method for calculating advantages through GAE-Lambda advantage calculation and rewards-to-go at the end of every epoch to return along with saved experiences.

4.3.2 Primary Execution Loop

By "Primary Execution Loop" or "main loop" I mean the central part of agent that handles initialisation, acting and learning. While the subsection 4.3.1 looks into the "separate cogs of the machine" this subsection will look at the basic use case of Spinning Up based agent as a whole.

The main loop of the VPG and the PPO agent utilises provided parameter to initialise the required data:

- Initialisation of Logger and model saving
- Definition of Seed for reproducibility of randomness
- Instantiation of environment
- Synchronisation of parameters across environments
- Setup of Experience Buffer
- Definition of methods for computation of policy and value loss
- Initialisation of ADAM optimisers for policy and value functions
- Definition of a method for updating neural networks as specified in subsections 2.5 and 2.6
- Saving of current time
- Preparation of environment through reset method

After initialisation, the main loop begins. In a given number of epochs of a specified size, an agent interacts with the environment and saves experience accrued into the experience buffer. Once an agent fails or an epoch reaches its maximum size, an update is performed over the model. This repeats until all the epochs have finished. Refer to figure 4.1 for visual reference of the main loop. Update utilises principles defined in subsection 2.5 for VPG and subsection 2.6 for PPO agents.

Through the cooperation of all these parts Spinning Up agent internalises a representation of an environment and tends to group up all interactions with environments in one big epoch that anything outside of the agent can't interact with. This architectury goes contrary to the

idea of AIQ that requires an agent to know almost nothing of the environment and to accept only a single interaction from outside of the agent

4.4 Architecture of new AIQ agents

To create an implementation that would allow an agent to interact with an environment arriving from outside of itself step by step instead of learning in batches, I have designed the following architecture for Agents:

4.4.1 Utility folder

I have extracted the necessary files from the Spinning Up utility folder into a new folder called *SpinUtils* inside the agent folder. I have modified these utility files to better fit the purposes and environments of our experiment. I have removed all Tensorflow dependencies to remove this deprecated library from my tech stack and modified logging to better fit our needs through the removal of stdout output and creation of internalised log database that gets outputted to a file only on request from an agent (that contains a method that allows AIQ to request this log).

I have compared the core files of VPG and PPO and found them identical. I have extracted these classes into the utility folder for less clutter and named the new files *Spincore*. These files contained core methods defining neural networks behind the agents, the so called Actors (Policy networks) and Critics (Value networks). I have modified these Actor Critic methods to utilise information passed from AIQ *refmachine* instead of OpenAI Gym. Next, I have removed the support for Gaussian environment as the BF machine is Discrete in both observation and action and thus always utilises Categorical actor.

I have also found out that VPG and PPO share not only core file but also their Buffer classes. As they are identical, I have extracted and merged these classes from the agent files and created a new utility file inside the *SpinUtils* folder called *PolicyEnvBuffer*. The code of this new file is identical to the code of the class in agents, except for its name.

4.4.2 Core Agent Class

A class with the same name as its implemented agent (*VPG.py*, *PPO.py* is a child class from the original abstract class Agent. As a child class, any implement agents inherit basic information about a reference machine, such as the number of observations and actions possible and discount rate. To enable the change into class from function, all passed hyperparameters of an agent are also stored as class attributes to allow access from all parts of the class.

But parameters are not the only thing a child class inherits. *Agent* defines abstract classes they require from their successors. These methods are *___str___* that returns the full name of an agent, *perceive* that accepts observation and reward as data and returns an action of the agent and method *reset* that reinitialises the agent and provides a clean instance of the agent. Reset not only reinitialises information about agent, but also logging information. As mentioned in 4.2.4 we require the ability to reset purely agent without restarting run logs. For this reason, all agent initialisation processes were moved to a separate method *setup_agent* that can be called from other places, thus allowing for the reinitialisation of an agent without touching any logging data. This setup specifically resets all inner counters of an agent and generates and saves a new instance of Actor-Critic module, buffer class and Policy and Value ADAM optimisers to their class attribute.

In this new implementation, the main loop of an agent is moved to the *perceive* method. This method accepts observation and reward from AIQ, and transforms it into necessary format that is a one-hot encoded tensor for observation and float for reward. Passes this information to the step method of the agent's instance of the Actor-Critic module. This method returns the predicted action with the best possible future return, along with the expected value and logarithm of the policy. These pieces of information, along with observation and reward, get saved to the experience buffer.

As the environment doesn't give any information about how many interactions have already happened, these new agents have to keep their own counter. I have implemented this counter through a new integer attribute of agent's class called *epoch_step*. After a step, the interaction counter is checked against the *steps_per_epoch* parameter to check if the policy should be updated. Either not enough experiences were collected, the interaction counter is incremented, and the action generated from the agent gets returned to AIQ to wait for another interaction, or it is time for training. To allow training of an agent, methods for acquiring information from acquired data, methods *compute_loss_pi* and *compute_loss_v* were transplanted in their entirety from the original Spinning Up implementation along with *Update* method. As these are the methods that define how the agent behaves, the only changes modified from the original implementation were replacing variables with class attributes and small change in logging data. With the core of an agent practically identical to the core of a tested and acknowledged AI package, there should be little doubt about the reliability of the implementation of these agents. Once networks are trained, the increment counter gets reset back to zero, and the action gets returned to AIQ.

As PPO is only a more advanced version of VPG, the code of both agents is mostly identical. The only difference between these two agents rests in the parameters passed to the agent class and the methods handling training of the networks. For PPO training methods *compute_loss_pi*, *compute_loss_v*, and *update* I have once again used the code of Spinning Up to ensure the correct behaviour of the agents.

5. Evaluation of implemented agents

In chapter 4, a description of how I implemented new agents can be found. With the agents implemented, it is time to evaluate their capabilities using the AIQ test.

Like many others, new agents have numerous hyperparameters that must be tuned to achieve the best results, which takes significant time and computational power. Thankfully the source code of (Achiam, 2018) that was used to implement new agents contains default values that can be used to achieve preliminary testing. Some parameters had to be modified, and a new parameter Steps_per_Epoch (SPE) was added as a replacement. As this is a new or, more specifically, modified hyperparameter, it lacks a default value, so finding one will be the primary goal of this chapter.

While searching for a great and acceptable range of values for this parameter in subsection 5.4.1, a range of data will be acquired, which will be used for additional analyses in this thesis to develop a preliminary understanding of how VPG and PPO compare in subsection 5.4.2 and how new agents compare to original ones in subsection 5.4.3 . Further analysis is beyond the scope of this thesis and will be continued as part of the AGIEva research.

5.1 Experiment Preparation

Before one can begin any experiment, there are things that one always has to specify: The goal of the experiment, the configuration of machines and programs and any additional tools needed.

Agent parameters: For this experiment, we are interested in testing agents PPO and VPG with multiple different values of Steps_per_Epoch ranging from extremely small value of ten to extremely large value of five thousand. The range of values chosen is:

[10, 50, 100, 500, 1000, 5000]

This range covers a wide range of possible agent behaviours. The other parameters visible in 5.1 correspond to default values defined in the (Achiam, 2018) implementation of agents. For each parameter's specific meaning, please refer to subsections 2.5 and 2.6.

Agent	t_pi_iters	t_v_iters	gamma	pi_lr	vf_lr	clip_ratio	target_kl
VPG	(1)[1]	80	0.99	0.0003	0.001		
PPO	80	80	0.99	0.0003	0.001	0.2	0.01

Table 5.1: Default values of parameters of implemented agents

Set of Environment Programs: After defining the hyperparameters of agents we wish to test, we need to define the environment programs the AIQ test will run on. For this, a new set of environment programs was generated using the updated internal generator of samples with the following command:

```
BF_sampler.py -s 200000 -r BF,5 \
\-\-improved_optimization \-\-improved_discriminativeness
```

By using this command, the generated set of samples has these settings:

- **Number of samples:** 200 000
- **Number of symbols:** 5
- **Optimisation:** Uses additional patterns for recognising pointless code in environmental programs
- **Discriminativeness:** Removes environment programs without discriminative power

For generation of these samples, settings used in (Vadinský, 2018b) are utilised.

Hardware & Software: The next step is preparing computers on which these tests will run. The MetaCentrum grid computing service provided by the e-INFRA CZ project (ID:90140) was utilised for calculation purposes. In provided hardware, a conda (Anaconda, Inc., 2021) environment was prepared to be used for testing. File in the format Yaml describing this environment can be found in provided files. For the specific location of the Yaml file, refer to the structure in appendix A.

Initialisation scripts: To ease my interaction with *Metacentrum*, I have created a Bash Script template of values for each agent parameter. When this script is filled in and executed, it generates a batch processing request based on a qsub configuration script file for each combination of values from provided lists of parameters. The preparation of various configurations becomes much more manageable through these two scripts. Details of the provided script template files and their usage will be provided in B

Configuration of AIQ: The final step was defining parameters to use when running AIQ. The code to start AIQ in the batch processing template file:

```
python AIQ.py --log --verbose_log_el --save_samples
--log_agent_failures -t "${threads}" -r "${_machine}"
-a "${_agent}" -l "${_episode}" -s " ${_sample}"
```

This code does the following:

- **python AIQ.py** - Start of program AIQ through python.
- **log** - Basic logging of sample interaction.
- **verbose_log_el** - Intermediary logging for every thousand steps over all samples.
- **save_samples** - Saves used programs to file, superseded by *log_agents_failures*.
- **log_agents_failures** - Information, if an agent failed during sample interaction, is added to a basic log and exports the internal log of an agent in case of agent failure.

- **-t** - Number of threads to run the program on - Passed from batch processing.
- **-r** - Reference machine that AIQ will be run on - BF - set in the template file.
- **-a** - Agent that will be run on AIQ - generated in batch generator and passed to configuration template as a parameter.
- **-l** - Number of interactions on every sample - 100 000 - set in the template file.
- **-s** - Total number of samples run - 10 000 - set in the template file.

5.2 Hypotheses

Before starting any experiment, there is a need to define hypotheses to test. Here we have three questions. How to configure new parameter of our agents, support AIQ as a valid evaluation by comparing VPG and PPO relation to their benchmark values and to compare newly implemented agents against the originally implemented ones.

5.2.1 Analysis of parameter steps_per_epoch

As noted at the beginning of chapter 5, due to changes made during the implementation of agents in the AIQ test, we lack default configuration for the parameter steps_per_epoch defining how many steps the agent takes between each update of policies. VPG and PPO are on-policy agents, meaning they can only learn from experiences gained during an epoch utilising current policy. This can lead to agents being unable to find any pattern in environments that return rewards based on actions done in distant history when the value of tested parameter Steps_per_Epoch is too small. Additionally, in environments with a limited number of steps, like AIQ test, this parameter defines how many training iterations are done over each policy. Too large of a training step can lead to insufficient training steps over policy. With not enough training steps, the policy will not converge before the end of testing and will return a much lower AIQ value. Based on these properties, I formulate the following hypothesis:

- By investigating the relationship between the learning speed, as determined by the parameter Steps_per_epoch, and the corresponding AIQ scores obtained from the AIQ test with an episode length of 100 000 steps, intriguing groups can be identified for further exploration. Particularly, focusing on the top 5%, 10%, and 20% of the best achieved AIQ scores might help in uncovering noteworthy patterns and insights in future research.

5.2.2 Comparison of VPG and PPO over default configuration

We can compare the AIQ score of newly implemented VPG and PPO agents by further analysing the data acquired. Through this analysis, we compare these agents against each other to find out if their results are similar to scores achieved in other experiments done in (Schulman; Wolski, et al., 2017) or benchmarks over Mujoco and ATARI environments in (Achiam, 2018). In these results, PPO performed better in all tested environments. By checking if the data acquired is similar to mentioned benchmarks we can further prove the validity of the AIQ test as an evaluation mechanism. For this analysis, we can formulate the following hypotheses.

- H0a: In comparing pairs of agents with the same parameter configuration SPE, there will be no significant difference between achieved AIQ scores of agents VPG and PPO.
- H1a: In comparing pairs of agents with the same parameter configuration SPE, the VPG agent will achieve significantly Lower AIQ scores than the PPO agent.

- H0b: In comparing pairs of agents with the same parameter configuration SPE, there will be no significant difference between the learning speed of agents VPG and PPO.
- H1b: In comparing pairs of agents with the same parameter configuration SPE, The VPG agent will learn significantly slower than the PPO agent.

5.2.3 Comparison of newly implemented agents with original over default configuration

Finally, we can compare how VPG and PPO compare against originally implemented agents of Q_0, Q_l, HLQ_l and freq. Comparing new agents with original agents should be an essential part of implementing new agents. However, to thoroughly compare multiple agents, it would be necessary to do a complete grid search of the new agent's parameters, which is far beyond the scope of this thesis. Instead, this section will briefly compare the tested agent's configurations that achieved the best total AIQ values in our testing. A simple hypothesis will suffice for this comparison.

- H0: There is no difference in the final AIQ scores among the different agents.
- H1: The final AIQ score of at least one agent is significantly higher than the mean final AIQ scores of the other agents.

5.3 Results

After all experiments are run, we are left with the data. After processing this data with the help of Python scripts, we acquire AIQ values after every thousand interactions with environments calculated from all samples. The entire table has been added to an appendix, but a brief look at AIQ values at 1000th, 3000th, 10 000th, 30 000th and 100 000th interaction can be found in 5.2 for VPG and in 5.3 for PPO.

VPG	Average AIQ at given interaction step				
Step Size	1000	3000	10000	30000	100 000
10	8.1	25.6	47.3	55.5	58.6
50	4.0	11.3	30.9	50.9	62.2
100	2.1	6.6	22.0	45.6	60.2
500	-0.3	0.6	4.2	15.3	44.0
1000	-0.1	0.4	2.3	7.7	28.5
5000	-0.4	-0.4	-0.1	1.1	4.8

Table 5.2: AIQ values for agent VPG on default parameter values

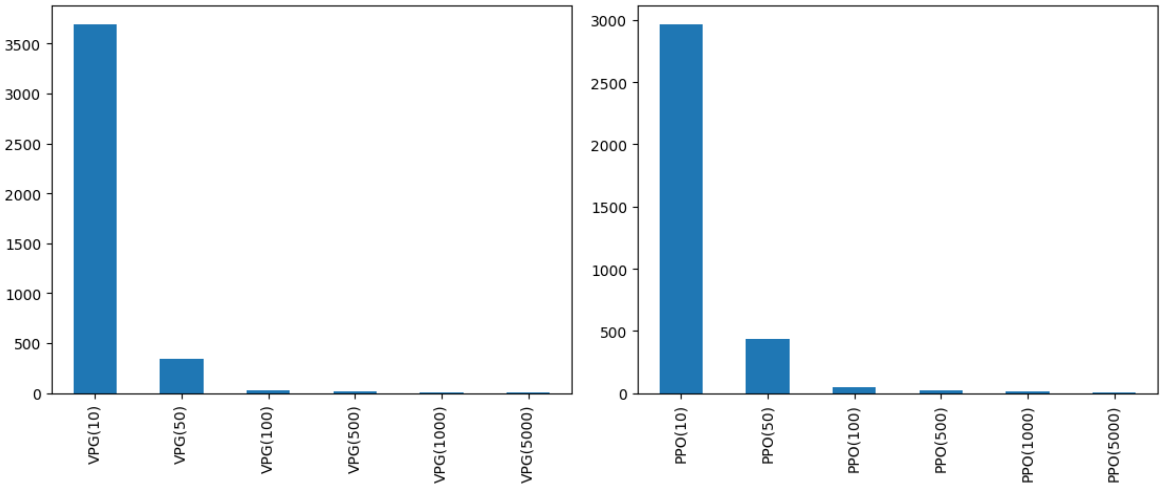
Values for PPO agent on step size ten have taken too long to calculate and had to be terminated before completion after eight days. Another run has been tried, but even after an extended time limit of 2 weeks the run didn't finish, and all the data acquired somehow disappeared from the computation servers. It was not further retested due to many agent failures, as seen in the graph 5.1 and significant processing time and power required.

PPO	Average AIQ at given interaction step				
Step Size	1000	3000	10000	30000	100 000
10					
50	28.3	49.1	57.0	61.7	64.6
100	21.4	45.5	55.0	60.5	63.9
500	3.4	23.5	45.3	59.4	64.5
1000	-0.3	11.1	30.9	54.1	63.6
5000	1.4	1.3	5.5	21.0	49.6

Table 5.3: AIQ values for agent PPO on default parameter values

During their testing, agents have managed to stumble on NaNs and fail. These failures have been logged, and their number can be seen in Figure 5.1.

When reading this graph, it is necessary to remember that each sample is run first with normal and then inverted parameters, so the number of failed runs is not out of ten thousand runs but out of twenty thousand runs. We can use percentages in bar graph 5.2 to visualise better how often an agent failed.



(a) VPG - Total failed sample runs

(b) PPO - Total failed sample runs

Figure 5.1: Graphs with total amount of failed sample runs

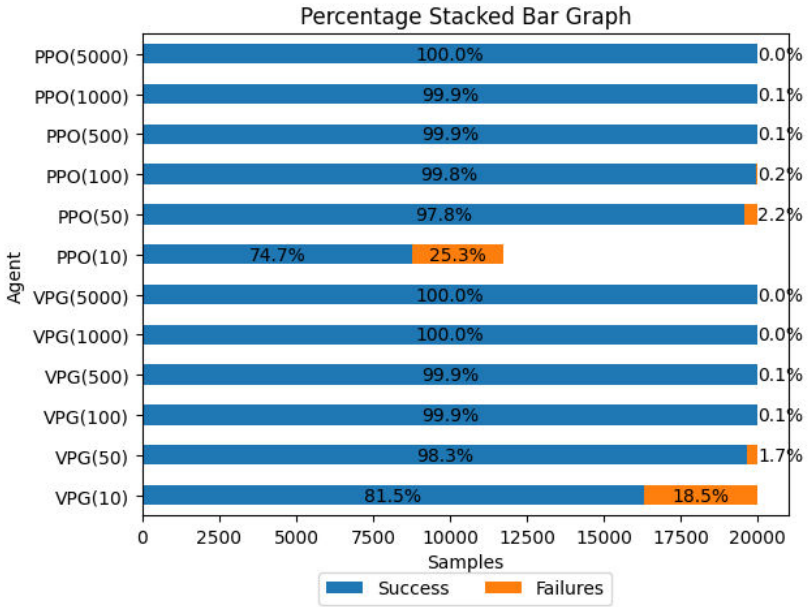


Figure 5.2: Graphs with percentage amount of failed and successful sample runs

It can be seen that lower the SPE parameter, the higher amount of failures an agent has, probably due to SPE influencing the size of the experience buffer and how many actions an agent can see. While deeper exploration of specific reasons for failure and ways to mitigate these failures is beyond the scope of this thesis and one of possible further research paths, it is important information to notice as too many failures lead to lower AIQ results and an unstable graph curve.

5.4 Analyses

With data acquired, it is time to analyse it. In section 5.4, three groups of hypotheses were formed. In this section, a subsection with an in-depth analysis will be created for each of the groups. In subsection 5.4.1, data will be evaluated to allow finding a good range of values for configuration of the parameter steps_per_epoch. In subsection 5.4.2 acquired data will be used to evaluate if newly implemented agents VPG and PPO have similar results against each other in AIQ test as they do in their benchmarks. Finally subsection 5.4.3 will look into how VPG and PPO compare against originally implemented agents Q_0, Q_l, HLQ_l and freq.

5.4.1 Comparison of VPG and PPO over default configuration

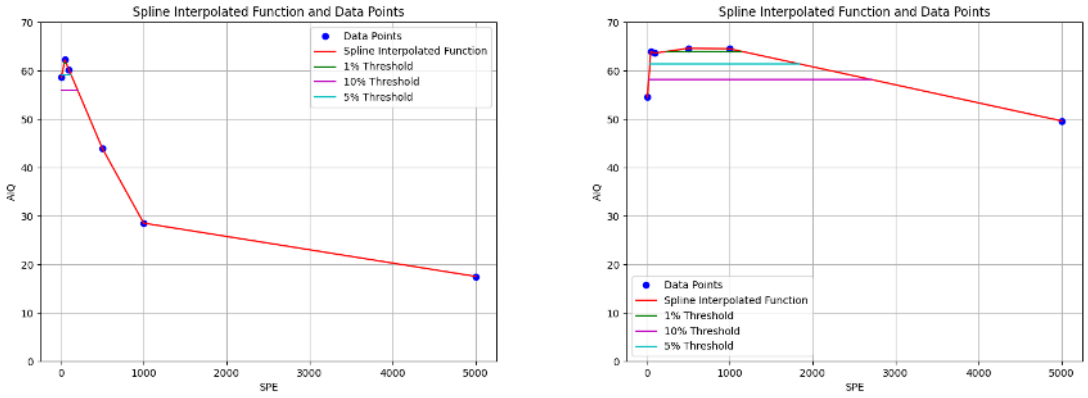
As can be seen from proposed hypothesis, we are interested in values of SPE that have the following:

- **Final AIQ achieved:** Highest final AIQ score achieved in the given number of interactions.
- **Speed of learning:** Highest rate of achieving AIQ score in short time.

Other attributes could be analysed, like time taken or calculation resources used, but we are currently interested mainly in AIQ values.

Final AIQ score achieved With this metric, we are interested in finding ranges of parameter SPE that contains the highest achieved values of AIQ score at 100 000 interactions. By using the current highest achieved AIQ as a maximum, I look for the range of SPE that should return the top 5%, 10% and 20% AIQ score of highest scoring configuration with default parameters to define ranges for the best, great and testable configurations. As this is the first rough exploration of this parameter, I believe that if we utilised lower percentages, we would miss a lot of possible interesting configurations. We are interested in finding these ranges to assist in choosing interesting parameter values for future testing.

To find a rough estimate of these "interesting" ranges, an interpolation of data has been done over the acquired data. While preparing interpolation, I have found that there is not enough data to create sensible quadratic or cubic interpolation, so the first analysis of this parameter will be done with linear interpolation. With the help of Python, I have interpolated the graphs 5.3a and 5.3b. From these interpolations I have found interesting ranges that can be found in table 5.4.



(a) VPG - Linear interpolation of AIQ results over SPE configurations

(b) PPO - Linear interpolation of AIQ results over SPE configurations

Figure 5.3: Graphs visualising linear interpolation calculated from acquired AIQ on tested SPE configurations

Top percentage	**5%**	**10%**	**20%**
PPO	<40,1840>	<26,48) ; (1840,2707>	<10,26) ; (2707,4441>
VPG	<16,127>	<10,16) ; (127,204>	(204,357>

Table 5.4: Acquired ranges of SPE parameter in specific percentages of best AIQ achieved

Speed of learning can be found by analysing the verticality of a graph describing AIQ acquired per environment interactions. The more vertical the curve of AIQ is, the faster the agent learns.

The first agent to analyse is the VPG in 5.4. As it runs only a single training step over policy every update, it learns slowly. It needs as many updates as possible, which means that for performance, the lower parameter value of Steps_per_Epoch are ideal. However, as seen in 5.1a, too low of a value for this parameter can lead to instability. So even if the parameter value of ten has the highest initial training speed, its instability starts projecting to the AIQ value curve from twenty thousand interactions. Too high of a value might have no instability but does not achieve enough training steps necessary to bring acceptable training speed. Both values of fifty and hundred have similar training speeds, with fifty having higher training speed but also higher instability, and hundred SPE being slower but more stable.

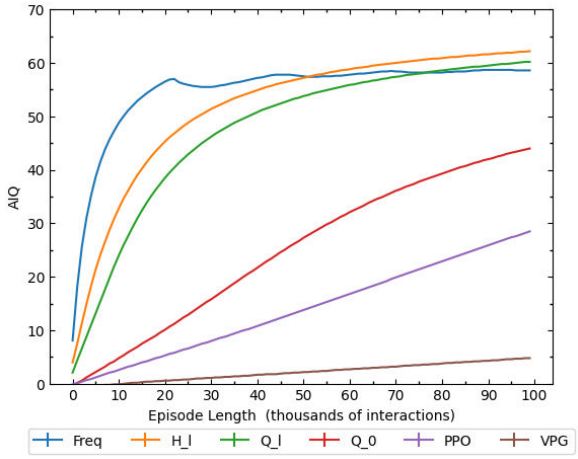


Figure 5.4: VPG - Graph of AIQ per thousands of interactions

Next to analyse is the PPO agent in 5.5. Like with VPG, it can be seen that too short of a SPE parameter leads to training instability. With PPO utilising multiple training steps every update, it projects to graph already in the first few updates. With multiple training steps per update, PPO can also achieve enough training iterations over policy for acceptable AIQ values even for larger SPE like five hundred or a thousand. Nevertheless, even if these larger parameter values manage to converge to high AIQ amounts in the total timesteps given, they are still way slower in training than smaller values like fifty or hundred. SPE of fifty is slightly unstable but already achieves AIQ of over thirty at first thousand steps while one hundred starts around 20 it continues learning at a similar speed as fifty. SPE of five thousand is too slow in learning, even if it is quite interesting that at this size, it achieves a speed of training comparable to VPG of five hundred steps per epoch.

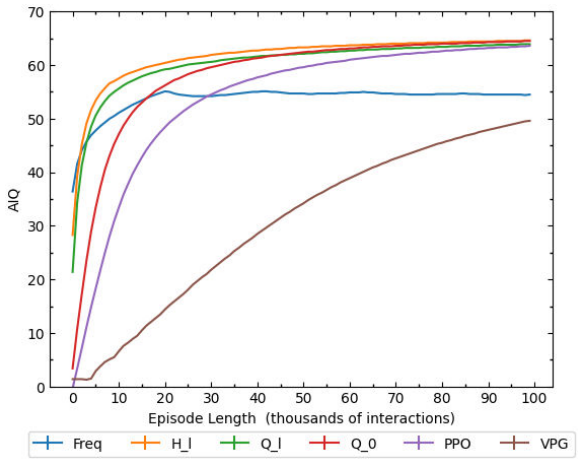


Figure 5.5: PPO - Graph of AIQ per thousands of interactions

To not only use visual analysis, we can also use the Area Under Curve to check learning speed of an agent. To acquire these values I have used the Simpson's rule "*scipy.integrate*" package.

SPE	10	50	100	500	1000	5000
VPG	5441.08	5131.48	4714.94	2470.08	1374.39	212.70
PPO	5318.38	6102.47	5977.23	5756.87	5307.15	3006.89

Table 5.5: Area Under Curve of VPG and PPO AIQ scores acquired through Simpson's rule

To once again use the 5%, 10% and 20% grouping we need to find the highest value of each agent. In this case, it is the value at SPE parameter of 10 for VPG and 50 for PPO. We will calculate the edge value for each group which will be used to divide the various configurations into groups according to learning speed.

	Best	5%	10%	20%
VPG	5441,08	5169,03	4896,97	4352,864
PPO	6102,47	5797,35	5492,22	4881,976

Table 5.6: Edge values for top groups of SPE configuration of various agents

SPE	10	50	100	500	1000	5000
VPG	Best	Top 10%	Top 20%	Below 20%	Below 20%	Below 20%
PPO	Top 20%	Best	Top 5%	Top 10%	Top 20%	Below 20%

Table 5.7: Grouping of SPE configurations of VPG and PPO according to top AUC percentages

With this, a rough approximation of interesting ranges of SPE parameter values according to the final achieved AIQ across various SPE configurations for each agent has been defined. To further improve the analysis, an analysis over learning speed is done. With this analysis we created another grouping of SPE configurations according to their achieved Area Under Curve.

5.4.2 Comparison of VPG and PPO over default configuration

After providing a rough estimate for suitable configurations of SPE parameter, the acquired data can be used for comparison of the new agents between each other. Thanks to having access to existing benchmarks across many various environments, we can compare acquired results against expectations from benchmarks to further prove that the implementation of new agents is correct and that AIQ is a valid test for RL agents.

This subsection will compare the following parts of the VPG and PPO:

A comparison of agent's results over the whole run: To achieve this comparison, every tested configuration of SPE parameter of PPO will be paired against the results of VPG with the same SPE configurations. Next through utilisation of statistical methods on the acquired AIQ results of every thousand steps up to a hundred thousand steps, the assumption that PPO is significantly better than VPG will be checked.

Comparison of the learning speed of two agents: A set of graphs plotting the learning speed of agents of the same configuration will be created, and their curves will be compared against each other. Next the Area Under Curve value will be computed and utilised for comparison.

Full run comparisons

Our hypothesis assumes that PPO will be better than VPG. To check if this assumption is correct, we need to use statistical methods. One type of these methods that are often used to compare two groups of values are the T-tests, more specifically, either a paired T-test or a two-sample T-test.

T-Test: A paired T-test is usually used to compare one group of individuals in two different sets of circumstances, for example, before and after an intervention. A two-sample T-test is used when you have two independent groups that you need to compare against each other.

Out of these two, the two-sample T-test fits our data sets better, so it should be the one that we would go for. Before one can utilise T-tests, some prerequisites are required to be checked. A T-test expects:

- Independent observation
- Normally distributed data
- Homogenity of Variances

Independent Observation expects that one group's values do not affect others' values. This is fulfilled in our data as final AIQ values across different tests do not affect each other.

Normally Distributed Data expects that the values in each group should follow a normal distribution. To check for this, we can utilise a statistical test. A Shapiro-Wilk test is one of the tests that can be used to check for the normality of a distribution. After running this test with the help of the scipy package, I have found that the p-value is lower than the significance value α of 0.05, so our assumption of normality is rejected

Homogenity of Variances expects that two groups being compared should have an approximately equal variance of values. A Levine's test can be utilised to check whether data has an equal variance. After acquiring the p-values through the help of the scipy package, we learn that, once again, the p-value is lower than the significance level $\alpha = 0{,}05$, and our assumption of equal variance is rejected.

By not fulfilling two out of three prerequisites of a T-test we have to look for a different more robust test that allows for comparison of two different groups of agents.

Mann-Whitney U test One of those tests is the Mann-Whitney U (MW-U) test. Non-parametric test of null hypothesis, usually used to determine if there is a difference in distributions, can also be used to determine if one distribution is greater than another. Compared to the T-tests, MW-U test assumptions only require that the sample drawn from the population is random, independent between each group and that measurements are at least ordinal. All of these assumptions are fulfilled, and the MW-U test can be used to compare our agents.

I have once again utilised the "*Scipy.Stats*" python package to utilise the Mann-Whitney U test. This test has been run through the command

```
statistic, p_value = mannwhitneyu(
    agent1_scores, agent2_scores, alternative='less'
)
```

Where *agent1_scores* are the VPG AIQ values every thousand steps, and *agent2_scores* are the PPO AIQ values every thousand steps. The parameter alternative defines what kind of assumption the test should check in the hypothesis. By specifying the value *less*, the test checks for the hypothesis that VPG has significantly lower overall AIQ values than PPO. By running data of our agents at the same SPE configurations through the MW-U test, we have found that our assumption may be correct. The Acquired P-Values are lower than the significance level $\alpha = 0{,}05$ at all tested configurations of SPE, which means we can reject the null hypothesis of the VPG and PPO agents having no significant difference in their achieved AIQ values at tested configurations of hyperparameters. For specific P-Values check the table 5.8

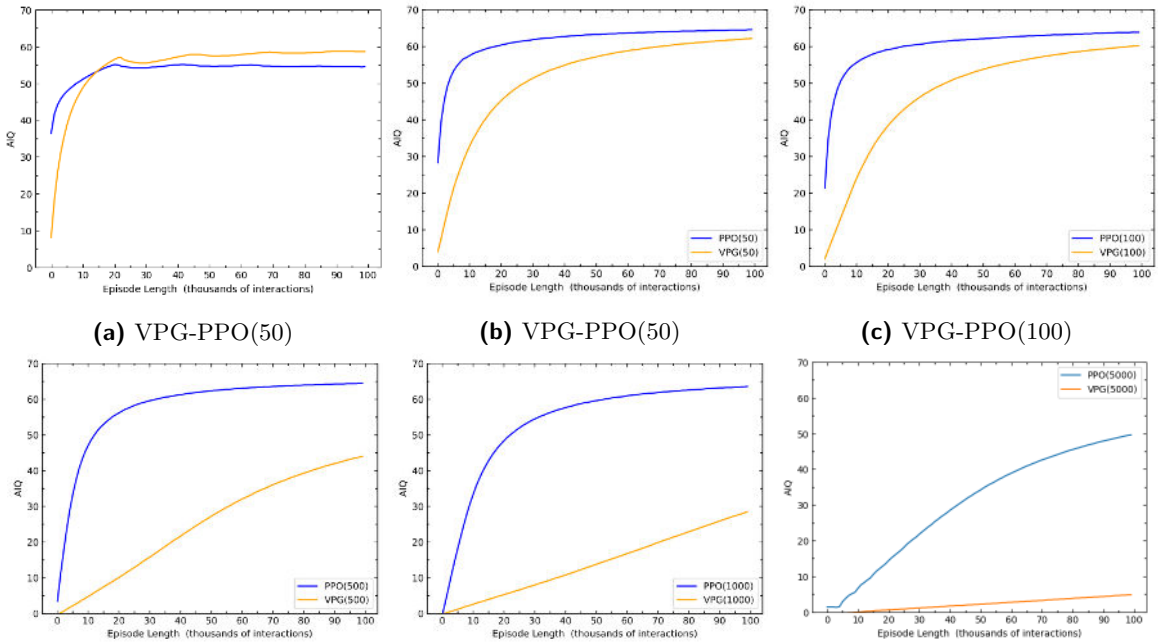
SPE	50	100	500	1000	5000
P_Value	1.61e-20	3.69e-23	2.38e-29	3.56e-29	7.65e-30
U-Statistic	1230.0	971.5	421.5	436.0	380.5

Table 5.8: Results of Mann-Whitney U test on AIQ acquired every thousand steps through SPE configurations of VPG and PPO agent

The SPE configuration of 10 was skipped due to the lack of finished full data of ten thousand samples from the PPO agent and a large number of agents' failures. Other than that, these findings support the alternative hypothesis that agent VPG has significantly lower AIQ values than agent PPO.

Learning speed

To look further into how VPG achieves a lower AIQ score than PPO, we first visually compared a plotted graph to guess how agents compare before calculating the Area Under Curve value for each graph and comparing these values.



(a) VPG-PPO(50) **(b)** VPG-PPO(50) **(c)** VPG-PPO(100)

(d) VPG-PPO(500) **(e)** VPG-PPO(1000) **(f)** VPG-PPO(5000)

Figure 5.6: Graphs comparing VPG and PPO training speeds at various configurations of parameter Steps_per_Epoch

From 5.6 it can be seen that PPO has a steeper training curve than VPG in all configurations. What is also interesting is that the curve of VPG is almost flattening with higher values of parameter SPE, contrary to the parameter PPO which is still capable of handling higher SPE values with a fairly high training curve. There is also the graph of AIQ at SPE configuration of 10, but with that one, we must remember that the PPO didn't finish the full test of 10 000 samples, and many of their samples had at least one agent failure during their run. This massively impacted its performance and allowed VPG to catch up to it.

To quantify these values, the use of the Area Under Curve (AUC) of the agents is once again required. For better readability, the table from 5.4.1 has been re-inserted

SPE	**10**	**50**	**100**	**500**	**1000**	**5000**
VPG_Simp	5441.08	5131.48	4714.94	2470.08	1374.39	212.70
PPO_Simp	5318.38	6102.47	5977.23	5756.87	5307.15	3006.89

Table 5.9: Area Under Curve of VPG and PPO AIQ scores acquired through Simpson's rule

As could be seen from the Figure 5.6, AUC value of SPE parameter 10 are an extreme exception due to high amount of agent failures. All the other VPG AUC values are massively lower than the AUC values of the PPO agent. To check for significant differences, I ran a Mann-Whitney U test where I compared acquired AUC values to check if VPG learns significantly slower than PPO.

P_Value_Simpson	**Statistic_Simpson**
0.02	5.00

Table 5.10: Results of Mann_Whitney U test over acquired AUC values (Simpson's rule, Composite Trapezoidal rule)

As seen in table 5.10 The P-Value acquired from comparing the AUC of both agents is lower than significance $\alpha = 0{,}05$. This means we can reject the null hypothesis of agents in tested configurations not having significant differences in their learning speed. These results also support our alternative hypothesis of VPG being significantly slower than PPO in given configurations.

5.4.3 Comparison of newly implemented agents with original over default configuration

The last analysis focuses on comparing newly implemented agents against the original agents. For this analysis, I will first visually compare the learning curves of all agents before running a one-sided two-sample t-test for permutations of all agents to find out which agents have significantly greater final AIQ score than others.

Data for comparison

Results were required to be over the same sample set to compare original and new agents. Data from VPG and PPO was acquired from the experiment in 5.4.1. For original agents, (Vadinský, Unreleased) supplied results of chosen agents (Freq, Q_l, Q_0, HLQ_l) configurations over the same sample set as utilised in 5.4.1. The configurations with the best final AIQ results of original and new agents were compared at the 1000th, 3000th, 10 000th, 30 000th and 100 000th step. Simple agents like random or manual that return either always 0 or whatever user behind manual would achieve were left out as they do not provide interesting data for our purposes.

The chosen configurations with their parameters are the following:

Agent	**Parameters**
Freq	(0,07)
HLQ_l	(0; 0; 0,995; 0,01; 0,8)
Q_l (Q_0)	(0; 0; 0,5; 0,005; 0,95)
Q_l	(0; 0,5; 0,5; 0,005; 0,95)
VPG	(50; 80; 0.99; 0.0003; 0.001)
PPO	(50; 80; 80; 0.99; 0.0003; 0.001; 0.2; 0.01)

Table 5.11: Parameters configuration of chosen agents with best final AIQ results

For an explanation of these parameters, refer to the original implementation of AIQ (Legg; Veness, 2011a) for original agents and to subsubsections 2.5 and 2.6 for new agents implementation. As all agents are off-policy agents and can use any data collected at any point in training, they are more sample efficient than newly implemented on-policy agents VPG and PPO, which can only use data gained from the latest policy state. As such, it can be expected that new agents VPG and PPO will need more training steps than original off-policy agents and will be slower, however after converging, their results might be similar or better than original Q-learning agents and a lot better than Freq agents that always chooses the most rewarding action with exceptions for random exploratory choices.

These configurations have the following AIQ values in chosen interaction steps:

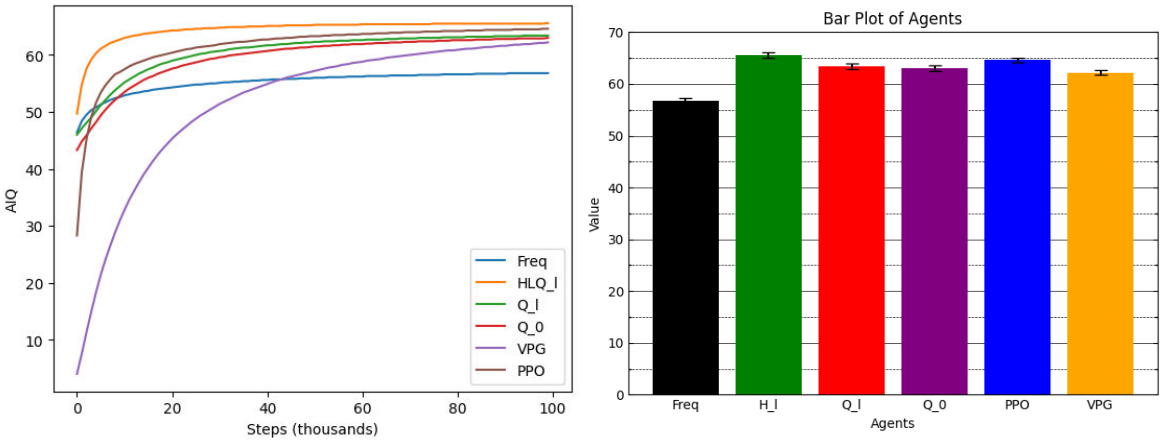
Steps	**Freq**	**HLQ_l**	**Q_0**	**Q_l**	**VPG**	**PPO**
1000	46,4	49,7	43,3	46,0	4,0	28,3
3000	49,5	57,6	45,9	48,1	11,3	45,2
10000	52,6	62,7	52,8	54,5	30,9	57,0
30000	55,0	64,7	59,4	60,6	50,9	61,7
100000	56,8	65,6	63,0	63,4	62,2	64,6

Table 5.12: AIQ values of agent configurations chosen for comparison

Comparison of agents

The training speed of VPG can be seen from 5.7 to be indeed the slowest of all agents at a single training step on the policy every training. PPO also learns slowly in the first few thousand steps. However, thanks to its capability to partly learn from more experiences than VPG due to its clipping and through performing eighty training steps in this configuration every update of policy, it starts to achieve similar AIQ values as rest of the original agents with the exception of HLQ_l between 3000 and 10 000 steps of interaction.

Through this simple observation, it could be proposed that the new *intelligence order relation* of agents would be as can be found in 5.13 with theoretically optimal AIXI above HLQ_l and random agent below Freq.



(a) AIQ graph of best configurations of tested agents **(b)** Final AIQ of best configurations of tested agents

Figure 5.7: Graphs of all tested agents at best acquired configuration

Agent	**HLQ_l**	**PPO**	**Q_l**	**Q_l (Q_0)**	**VPG**	**Freq**
Position	1th	2nd	3rd	4th	5th	6th

Table 5.13: *Intelligence Order Relation* of original and new agents

To further check this assumption, a one-sided (greater than hypothesis) two-sample t-test on all agent permutations can be run to check which agent is significantly better than other agents. The test is performed on every agent's final 100 000th AIQ test results. AIQ score is a weighted mean, and AIQ test also gives us a weighted standard deviation. As all agents' mean and standard deviation look comparable, we can assume equal variances for the t-test. We can use the following equation to calculate the twosample ttest.

Equations of weighted two_sample ttest:.

$$t_{value} = \frac{weighted\ mean1 - weighted\ mean2}{\sqrt{\frac{weighted\ standard\ deviation1^2}{n_1} + \frac{weighted\ standard\ deviation2^2}{n_2}}}$$

For acquiring T_Value where n is the number of samples 10 000 for both n1 and n2 and

$$p = 1 - cdf(t\_value, df)$$

For acquiring P_value where cdf is the cumulative distributive function calculated using python scipy function "*stats.t.cdf*" and df is the degree of freedom calculated by number of samples minus 1 $(10000 - 1)$

By running these tests and saving the acquired P_values into a matrix, we can check which agents are significantly greater than other agents by going through the row (and aren't significantly greater by going through the column). The resulting table 5.14 has coloured cells with values lower than significance level $\alpha = 0{,}05$. Due to extremely small values of some P_values, the calculation is not precise enough to return specific values and is counted as if it was almost 0.

	Freq	H_l	Q_l	Q_0	PPO	VPG
Freq		1	1	1	1	1
H_l	0		3,33E-11	1,89E-15	0,001156	0
Q_l	0	1		0,116899	0,999837	0,000115298
Q_0	0	1	0,883101		0,999999	0,006092517
PPO	0	0,998844	0,000163	5,20E-07		1,98E-14
VPG	0	1	0,999885	0,993907	1	

Table 5.14: P_Values from weighted two-sample t-test from all agent permutations

As can be seen, our assumption for intelligence order relation is supported by results of two-sample t-test with H_l being significantly greater than all other agents followed by PPO being significantly greater by all other than Hłand so on for Qł, Q_0, VPG and finished by Freq that has no other agent it is significantly better than. From this t-test, we can reject the null hypothesis of there being no significant difference between agents and, for now, accept the alternative hypothesis to be further confirmed in future research

5.5 Discussion

The goal of the analyses in this thesis was threefold. Find missing default parameters and use data acquired from the search for those parameters to make a preliminary rough comparison of VPG and PPO to prove similar behaviour to benchmarks and to find how new agents score compared to original agents.

During testing, some other interesting information appeared in the form of agent failures. One thing that could be noted is that the lower the value of SPE parameter became, the higher the number of player failures rose. This could be due to either having too small an experience buffer leading to the inability to notice larger behavioural patterns, or over-training and failing when advantage values lower too close to zero leading towards precision problems returning NaN. Performing any exploration beyond the surface overview of why such problems happen and how to fix them is out of this thesis's scope and possible future research.

5.5.1 Analysis of parameter Steps_per_Epoch

The first exploratory hypothesis focused on exploring a good range of values for a new parameter that did not contain default values in the original implementation of agents. The hypothesis proposed that a relationship between new parameter Steps_per_Epoch and achieved AIQ agent scores can be used to find intriguing groups of various success.

As the testing was done only on a relatively small set of SPE parameter values, a focus was on parameters that achieved the top 5, 10 and 20 percent of the best scoring configuration

across all tested agent configurations. By allowing higher percentage values, we achieve wider acceptable ranges to be used in future exploration, leading to more precise relationship prediction through interpolation (Ideally cubic or geometric) or curve fitting and perhaps even finding association rules with other hyperparameters.

While σ is usually used for such matters, the question of which standard deviation to use was hard to solve as, in our data, most of our standard deviations were fairly high, which would have led to too large groups of data and margins of error were too small which would have the opposite problem of too small groups of data.

While searching for acceptable ranges, I have decided not to focus on just tested configurations but to try to predict possible SPE parameter values that could belong to defined groups. For this, I have used linear interpolation, which allowed me to find more specific predictions towards interesting parameter configurations that can be found in table 5.4 with best values predicted to rest between SPE of 40 to 1840 for the PPO agent and 16 to 127 for the VPG agent.

Additionally I have looked into learning speeds of various configurations of SPE parameters between VPG and PPO and through calculation of Area Under Curve and calculating percentages of best achieved AUC of every agent I have created additional grouping for specific SPE parameter configurations. I have found that the best possible learning speed is SPE parameter of 10 for the VPG agent and 50 for the PPO agent. Next I have also found that while PPO only has one tested configuration of SPE that has learning speed below 20% of the highest achieved AUC value, half of the tested configurations of VPG belong to that grouping. For the rest of grouping please refer to table 5.7.

5.5.2 Comparison of VPG and PPO over default configuration

In the second analysis, we have utilised data acquired while searching for suitable values of Steps_per_Epoch to compare Vanilla Policy Gradient to Proximal Policy Optimisation. As PPO is a direct improvement to VPG, it can be expected that its performance will be better. Benchmarks agree with this as all tests in (Schulman; Wolski, et al., 2017) and (Achiam, 2018) tests over Mujoco and ATARI environments show PPO performs much better than VPG. Achieving similar results in the AIQ test could further prove that AIQ is a valid test for Various agents, including Policy Gradient type of agents.

The hypotheses created focused on two main aspects of an agent that interests us: achieved AIQ score and learning speed. By comparing all logged AIQ scores of every thousand steps between pairs of agents with the same SPE configurations using statistical methods, further support towards our first alternative hypothesis that VPG performs significantly worse than PPO could be provided. During testing assumptions of t-tests, I learned that normality and variance requirements are not fulfilled, so I had to use a more robust statistical test in the form of the Mann-Whitney U test. By performing this test on the hypothesis that VPG has a lesser AIQ score than PPO, we have achieved the p_value with extremely small amounts

below $1 \times e^{-20}$ for all configurations, which are all massively below significance level $\alpha = 0.05$. These findings can allow us to reject the null hypothesis of no significant difference between the two agents and support our alternative hypothesis of VPG achieving significantly lower AIQ than PPO.

The second hypothesis focused on Learning speed. Through first visually analysing graphs, we have found that PPO can handle changes in SPE way better than VPG, especially on very high parameter values. However, due to its learning iterations over policy, the lower the SPE value, the more time and processing power it takes for PPO to run. VPG, on the other hand, almost flattens out in the graph once SPE reaches values beyond one thousand. Across all graphs, it can be seen that the curve of the PPO agent is way steeper than the curve of the VPG agent. Sadly, just visual comparison cannot tell us anything about the significance in relations between agents. For this, I have calculated the Area Under Curve value to represent its learning speed instead through the utilisation of the Simpson's rule. Acquired AUC values were then compared using a Mann-Whitney U test across all configurations simultaneously. The resulting P_Value of 0,02 leads us to reject null hypothesis of both agents having no significant difference in learning speed and provides support for the alternative hypothesis H1b of agent VPG having significantly worse learning speed than agent PPO.

5.5.3 Comparison of newly implemented agents with original over default configuration

The final analysis focused on comparing new agents with the initially implemented ones. As PPO and VPG are on-policy agents, they can only learn from data acquired during the last run of the policy. This means that they are way less efficient with data and learn slower. Yet, they can be more stable than other off-policy agents and, in the end, should achieve a similar score to the original agents. In this comparison, we will only focus on the most important information - the Final achieved AIQ score, and learning speed will only be briefly examined over a simple graph.

From the graph 5.7a, it can be seen that both PPO and VPG start at lower AIQ values compared to the original agents. PPO, contrary to our expectations, achieves a higher learning curve than the Q, VPG and especially Freq agents. VPG, as assumed, learns slower than all but the simple Freq agent, barely catching up to other agents at the final step of 100 000thousand steps. From the shape of the graph, it might be possible to say that VPG did not fully converge at 100 000 steps yet and might achieve an even higher AIQ score if the test continued.

For the final AIQ score, all agents except the most simple Freq achieved between 60 and 65 AIQ scores. By utilising purely AIQ score, a preliminary Intelligence Order Relation can be constructed. Margins of error make some of these agent's possible AIQ scores overlap. To provide better support for creating a sequence of agents according to their intelligence, I perform a one-sided two-sample t-test on permutations of all agents, testing the hypothesis of

one agent having higher final AIQ than the second compared agent, utilising their AIQ score as weighted mean and their standard deviation value as their weighted variance (weighted standard deviation). By providing this set of tests, I created a table 5.14. Going by a row of this table for each agent allows us to see for which agents given agent has $p\_value < \alpha = 0.05$. Cells with such values are coloured green in the table, showing that the agent the row label shows are significantly greater than the agent labeled in that column. The null hypothesis of no significant difference between various agents can be rejected thanks to this. Our analyses support that HLQ_l is significantly better than all other agents. A sequence of achieved AIQ scores for all other agents could be created along with support for each agent being significantly better than all other agents below their position.

Conclusion

The goal of this thesis was to assess Policy optimisation agents using AIQ. This was fulfilled by choosing and implementing pure policy optimisation agents: Vanilla Policy Gradient and Proximal Policy Optimization, and implementing them into AIQ. This goal focused on testing different categories of agents than currently implemented. During implementation, this goal extended to bringing AIQ further in time to a more modern code foundation.

Summary of thesis's results

Through intensive research, the main history behind AIQ and Universal intelligence was introduced along with their main principles in chapter 1. Chapter 2 introduced the Reinforcement learning framework along with how agents using this framework are categorised. This introduction includes a small overview of some of the newer RL agents with basic explanations of their principles.

As both the original (Legg; Veness, 2011b) and improved (Vadinský, 2019) implementation of AIQ agent was programmed in deprecated Python 2, a significant part of the thesis became updating the code of AIQ to a more modern version of Python in chapter 3 along with implementing compatibility for running AIQ on Windows. During the implementation of new agents in section 4, it was found that some parameters had to be replaced to be compatible with AIQ and finding valid configurations for this parameter over the default settings of the remaining parameters became one of the main focuses of the thesis in chapter 5.

With the data gained during the search for valid configurations of missing default parameter values, the new agents were compared, and the hypothesis that in the current configuration of AIQ and default configuration of agents, PPO is better than VPG was proven. Finally, the new agents were briefly compared to the originally implemented agents over default configurations of new agents. A grid search over the parameters of new agents is required to accurately compare new agents with the initially implemented ones. Due to this being both computationally expensive and time intensive, it was deemed above this thesis's scope.

Contributions to the field

As the main contribution of this work, I would define the update of the AIQ algorithm to a newer code base. This allows for more research over AIQ and is already actively utilised in research project AGIEVA and several other in-progress works.

The following significant contribution is the implementation of new agents into the AIQ test. As these agents are of different categories and principles than initially implemented, it allows

for more research to be done and many new hypotheses to be formed and analysed in future testing. Additionally, the steps taken in this work can be used as inspiration for implementing additional agents into AIQ.

Another important contribution is defining new good and interesting values of parameters that were changed during implementation. Finding a good range of values allows for better specification of parameters during future research. It was also proven that the relation between VPG and PPO is similar to the ATARI and MUJOCO benchmarks in (Achiam, 2018), further demonstrating the validity of AIQ testing.

While a brief comparison between the initially implemented and new agents was made, much more research is required. My work in this comparison is purely a small contribution to be extended in future.

Usable results of this work

The main practical results of this work are:

- The update of AIQ code to Python 3.8 was released as open-source code, allowing for more research into AIQ on the newer code base.
- Additional improvements in AIQ for use in research project AGIEVA
- Implementation of new agents VPG and PPO for further research in project AGIEVA
- Results of tests done over the default parameter configuration of implemented agents as defined in (Achiam, 2018) for further analysis in research project AGIEVA
- Helpful scripts for batch-processing in Metacentrum for easier further testing in research project AGIEVA

Aside from practical results, this work also brings an overview of the theory behind the evaluation of Universal Intelligence and its approximation, along with an introduction to Reinforcement Learning, categorization of its agents and a brief overview of the main principles behind some of the newer agents. Finally, a valid range of possible values for parameter *steps_per_epoch* of newly implemented agents is defined for further testing.

Possible future research

This thesis has many possible avenues for future expansion. One of the possible categories in future expansion is further research into agents implemented in this thesis. Research such as:

- Comparison with originally implemented agents over the same values of shared parameters.
- Full grid search of possible parameters in AIQ agent.
- Improvements on the stability of newly implemented agents.

Next, there is the possibility of further research into the implementation of AIQ itself:

- Changing the behaviour of AIQ testing in case of agent failure.
- Implementation of additional agents for future testing.
- Implementing compatibility with modern environment frameworks like OpenAI Gym (Brockman et al., 2016) for easier implementation of new agents.
- Further refactoring of the AIQ code to achieve a cleaner and faster code base.

Possibilities proposed here are only a small sample of possible research that could continue from this work.

Bibliography

ACHIAM, Joshua, 2018. Spinning Up in Deep Reinforcement Learning.

ANACONDA, INC., 2021. *Conda documentation*. Austin, TX. Available also from: `https://docs.conda.io/projects/conda/en/latest/`. Version 4.10.1.

ANDRYCHOWICZ, Marcin; WOLSKI, Filip; RAY, Alex; SCHNEIDER, Jonas; FONG, Rachel; WELINDER, Peter; MCGREW, Bob; TOBIN, Josh; ABBEEL, Pieter; ZAREMBA, Wojciech, 2017. Hindsight Experience Replay. *CoRR*. Vol. abs/1707.01495. Available from arXiv: `1707.01495`.

BARAZZA, Leonardo, 2019. *VPG-Pytorch*. Available also from: `https://github.com/lbarazza/VPG-PyTorch`.

BARTO, Andrew G.; SUTTON, Richard S.; ANDERSON, Charles W., 1983. NEURONLIKE ADAPTIVE ELEMENTS THAT CAN SOLVE DIFFICULT LEARNING CONTROL PROBLEMS. *IEEE transactions on systems, man, and cybernetics*. Vol. 13, no. 5, pp. 834–846. ISBN 0018-9472.

BELLEMARE, Marc G.; DABNEY, Will; MUNOS, Rémi, 2017. *A Distributional Perspective on Reinforcement Learning*. arXiv. Available from DOI: `10.48550/ARXIV.1707.06887`.

BRETT CANNON, 2013. *Porting Python 2 Code to Python 3*. Available also from: `https://docs.python.org/3/howto/pyporting.html`.

BROCKMAN, Greg; CHEUNG, Vicki; PETTERSSON, Ludwig; SCHNEIDER, Jonas; SCHULMAN, John; TANG, Jie; ZAREMBA, Wojciech, 2016. *OpenAI Gym*. Available from eprint: `arXiv:1606.01540`.

CONTRIBUTORS, The garage, 2019. *Garage: A toolkit for reproducible reinforcement learning research* [`https://github.com/rlworkgroup/garage`]. GitHub.

DESCARTES, RENÉ, 1637. *A Discourse on the Method of Correctly Conducting One's Reason and Seeking Truth in the Sciences*. OXFORD WORLD'S CLASSICS.

DHARIWAL, Prafulla; HESSE, Christopher; KLIMOV, Oleg; NICHOL, Alex; PLAPPERT, Matthias; RADFORD, Alec; SCHULMAN, John; SIDOR, Szymon; WU, Yuhuai; ZHOKHOV, Peter, 2017. *OpenAI Baselines* [`https://github.com/openai/baselines`]. GitHub.

DOWE, David L.; HÁJEK, Alan, 1998. A Non-Behavioural, Computational Extension to the Turing Test. In.

DUAN, Yan; CHEN, Xi; HOUTHOOFT, Rein; SCHULMAN, John; ABBEEL, Pieter, 2016. *Benchmarking Deep Reinforcement Learning for Continuous Control*. Available from arXiv: `1604.06778 [cs.LG]`.

FEINBERG, Vladimir; WAN, Alvin; STOICA, Ion; JORDAN, Michael I.; GONZALEZ, Joseph E.; LEVINE, Sergey, 2018. *Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning*. arXiv. Available from DOI: `10.48550/ARXIV.1803.00101`.

FOUNDATION, Farama, 2022. *Announcing The Farama Foundation The future of open source reinforcement learning*. Available also from: https://farama.org/Announcing-The-Farama-Foundation.

FUJIMOTO, Scott; HOOF, Herke van; MEGER, David, 2018. *Addressing Function Approximation Error in Actor-Critic Methods*. arXiv. Available from DOI: 10.48550/ARXIV.1802.09477.

GOERTZEL, B., 2015. Artificial General Intelligence. *Scholarpedia*. Vol. 10, no. 11, p. 31847. Available from DOI: 10.4249/scholarpedia.31847. revision #154015.

GOTTFREDSON, Linda S, 1997. Why g matters: The complexity of everyday life. *Intelligence*. Vol. 24, no. 1, pp. 79–132.

HA, David; SCHMIDHUBER, Jürgen, 2018. Recurrent World Models Facilitate Policy Evolution. In: *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., pp. 2451–2463. Available also from: https://papers.nips.cc/paper/7512-recurrent-world-models-facilitate-policy-evolution. https://worldmodels.github.io.

HAARNOJA, Tuomas; ZHOU, Aurick; ABBEEL, Pieter; LEVINE, Sergey, 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *CoRR*. Vol. abs/1801.01290. Available from arXiv: 1801.01290.

HARNAD, Stevan, 1991. Other bodies, other minds: A machine incarnation of an old philosophical problem. *Minds and Machines*. Vol. 1, pp. 43–54. Available also from: https://www.southampton.ac.uk/~harnad/Papers/Harnad/harnad91.otherminds.html.

HASSELT, Hado van; GUEZ, Arthur; SILVER, David, 2015. Deep Reinforcement Learning with Double Q-learning. *CoRR*. Vol. abs/1509.06461. Available from arXiv: 1509.06461.

HERNANDEZ-ORALLO, Jose, 1999. Beyond the Turing Test. *Journal of Logic, Language and Information*. Vol. 9. Available from DOI: 10.1023/A:1008367325700.

HERNÁNDEZ-ORALLO, José; DOWE, David L., 2010. Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*. Vol. 174, no. 18, pp. 1508–1539. ISSN 0004-3702. Available from DOI: https://doi.org/10.1016/j.artint.2010.09.006.

HEYLIGHEN, Francis, 1999. Advantages and limitations of formal expression. *Foundations of Science*. Vol. 4.

HIBBARD, Bill, 2009. Bias and No Free Lunch in Formal Measures of Intelligence. In: *Journal of Artificial General Intelligence*.

HILL, Ashley; RAFFIN, Antonin; ERNESTUS, Maximilian; GLEAVE, Adam; KANERVISTO, Anssi; TRAORE, Rene; DHARIWAL, Prafulla; HESSE, Christopher; KLIMOV, Oleg; NICHOL, Alex; PLAPPERT, Matthias; RADFORD, Alec; SCHULMAN, John; SIDOR, Szymon; WU, Yuhuai, 2018. *Stable Baselines* [https://github.com/hill-a/stable-baselines]. GitHub.

HUI, Jonathan, 2018. *RL — Proximal Policy Optimization (PPO) Explained*. 2018-09. Available also from: https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12.

HUTTER, Marcus, 2004. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer Science & Business Media.

HUTTER, Marcus; LEGG, Shane, 2008. Temporal Difference Updating without a Learning Rate. Available from DOI: 10.48550/ARXIV.0810.5631.

INSA-CABRERA, Javier; DOWE, David; ESPAÑA, Sergio; HERNÁNDEZ-LLOREDA, Victoria; HERNANDEZ-ORALLO, Jose, 2011. Comparing Humans and AI Agents. In: pp. 122–132. ISBN 978-3-642-22886-5. Available from DOI: 10.1007/978-3-642-22887-2_13.

KINGMA, Diederik P.; BA, Jimmy, 2014. *Adam: A Method for Stochastic Optimization*. arXiv. Available from DOI: 10.48550/ARXIV.1412.6980.

KOLMOGOROV, Andrei N, 1965. Three approaches to the quantitative definition ofinformation'. *Problems of information transmission*. Vol. 1, no. 1, pp. 1–7.

LEGG, Shane; HUTTER, Marcus, 2007. *Universal Intelligence: A Definition of Machine Intelligence*. Available from arXiv: 0712.3329 [cs.AI].

LEGG, Shane; VENESS, Joel, 2011a. *AIQ*. GitHub. Available also from: %5Curl%7Bhttps://github.com/mathemajician/AIQ%7D.

LEGG, Shane; VENESS, Joel, 2011b. *An Approximation of the Universal Intelligence Measure*. Available from arXiv: 1109.5951 [cs.AI].

LEIKE, Jan; HUTTER, Marcus, 2015. *Bad Universal Priors and Notions of Optimality*. Available from arXiv: 1510.04931 [cs.AI].

LILLICRAP, Timothy P.; HUNT, Jonathan J.; PRITZEL, Alexander; HEESS, Nicolas; EREZ, Tom; TASSA, Yuval; SILVER, David; WIERSTRA, Daan, 2015. *Continuous control with deep reinforcement learning*. arXiv. Available from DOI: 10.48550/ARXIV.1509.02971.

MAHONEY, Matthew V., 1999. Text Compression as a Test for Artificial Intelligence. In: *AAAI/IAAI*.

MARTÍN ABADI; ASHISH AGARWAL; PAUL BARHAM; EUGENE BREVDO; ZHIFENG CHEN; CRAIG CITRO; GREG S. CORRADO; ANDY DAVIS; JEFFREY DEAN; MATTHIEU DEVIN; SANJAY GHEMAWAT; IAN GOODFELLOW; ANDREW HARP; GEOFFREY IRVING; MICHAEL ISARD; JIA, Yangqing; RAFAL JOZEFOWICZ; LUKASZ KAISER; MANJUNATH KUDLUR; JOSH LEVENBERG; DANDELION MANÉ; RAJAT MONGA; SHERRY MOORE; DEREK MURRAY; CHRIS OLAH; MIKE SCHUSTER; JONATHON SHLENS; BENOIT STEINER; ILYA SUTSKEVER; KUNAL TALWAR; PAUL TUCKER; VINCENT VANHOUCKE; VIJAY VASUDEVAN; FERNANDA VIÉGAS; ORIOL VINYALS; PETE WARDEN; MARTIN WATTENBERG; MARTIN WICKE; YUAN YU; XIAOQIANG ZHENG, 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Available also from: https://www.tensorflow.org/. Software available from tensorflow.org.

MCCARTHY, John, 1998. What is artificial intelligence. *URL: http://www-formal. stanford. edu/jmc/whatisai. html*. Available also from: http://www.cis.umassd.edu/~ivalova/Spring08/cis412/Old/WHATISAI.PDF.

MESSAGE PASSING INTERFACE FORUM, 2021. *MPI: A Message-Passing Interface Standard Version 4.0*. Available also from: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

MNIH, Volodymyr; BADIA, Adrià Puigdomènech; MIRZA, Mehdi; GRAVES, Alex; LILLICRAP, Timothy P.; HARLEY, Tim; SILVER, David; KAVUKCUOGLU, Koray, 2016. Asynchronous Methods for Deep Reinforcement Learning. Available from DOI: 10.48550/ARXIV.1602.01783.

MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; GRAVES, Alex; ANTONOGLOU, Ioannis; WIERSTRA, Daan; RIEDMILLER, Martin, 2013. *Playing Atari with Deep Reinforcement Learning*. arXiv. Available from DOI: 10.48550/ARXIV.1312.5602.

MÜLLER, Urban, 1993. *BF Code Readme*. GitHub. Available also from: %5Curl%7Bhttps://gist.github.com/rdebath/0ca09ec0fdcf3f82478f#file-readme%7D.

NAGABANDI, Anusha; KAHN, Gregory; FEARING, Ronald S.; LEVINE, Sergey, 2017. *Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning*. arXiv. Available from DOI: 10.48550/ARXIV.1708.02596.

ONDŘEJ, Vadinský, 2018. Towards General Evaluation of Intelligent Systems: Lessons Learned from Reproducing AIQ Test Results. *Journal of Artificial General Intelligence*. Vol. 9, no. 1, pp. 1–54. Available also from: https://www.proquest.com/scholarly-journals/towards-general-evaluation-intelligent-systems/docview/2021797998/se-2. Copyright - Copyright De Gruyter Open Sp. z o.o. 2018; Poslední aktualizace - 2019-09-06.

OPENAI, 2020. *OpenAI Standardizes on PyTorch*. Available also from: https://openai.com/blog/openai-pytorch/.

PASZKE, Adam; GROSS, Sam; MASSA, Francisco; LERER, Adam; BRADBURY, James; CHANAN, Gregory; KILLEEN, Trevor; LIN, Zeming; GIMELSHEIN, Natalia; ANTIGA, Luca; DESMAISON, Alban; KOPF, Andreas; YANG, Edward; DEVITO, Zachary; RAISON, Martin; TEJANI, Alykhan; CHILAMKURTHY, Sasank; STEINER, Benoit; FANG, Lu; BAI, Junjie; CHINTALA, Soumith, 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., pp. 8024–8035. Available also from: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

RAFFIN, Antonin; HILL, Ashley; GLEAVE, Adam; KANERVISTO, Anssi; ERNESTUS, Maximilian; DORMANN, Noah, 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*. Vol. 22, no. 268, pp. 1–8. Available also from: http://jmlr.org/papers/v22/20-1364.html.

RICHTER, Oliver; WATTENHOFER, Roger, 2019. Quantile Regression Deep Reinforcement Learning. *CoRR*. Vol. abs/1906.11941. Available from arXiv: 1906.11941.

RJ, WILLIAMS, 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine learning*. Vol. 8, no. 3-4, pp. 229–256. ISBN 0885-6125.

SEARLE, John R, 1980. Minds, brains, and programs. *Behavioral and brain sciences*. Vol. 3, no. 3, pp. 417–424.

SEITA, Daniel, 2018. *Actor-Critic Methods: A3C and A2C*. Available also from: https://danieltakeshi.github.io/2018/06/28/a2c-a3c/.

SCHULMAN, John, 2016. *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs*. Available also from: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-217.html. PhD thesis. EECS Department, University of California, Berkeley.

SCHULMAN, John; LEVINE, Sergey; MORITZ, Philipp; JORDAN, Michael I.; ABBEEL, Pieter, 2015. *Trust Region Policy Optimization*. arXiv. Available from DOI: 10.48550/ARXIV.1502.05477.

SCHULMAN, John; MORITZ, Philipp; LEVINE, Sergey; JORDAN, Michael; ABBEEL, Pieter, 2015. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. arXiv. Available from DOI: 10.48550/ARXIV.1506.02438.

SCHULMAN, John; WOLSKI, Filip; DHARIWAL, Prafulla; RADFORD, Alec; KLIMOV, Oleg, 2017. *Proximal Policy Optimization Algorithms*. Available from arXiv: 1707.06347 [cs.LG].

SCHWEIZER, Paul, 2012. The Externalist Foundations of a Truly Total Turing Test. *Minds and Machines*. Vol. 22, pp. 191–212.

SILVER, David; HUBERT, Thomas; SCHRITTWIESER, Julian; ANTONOGLOU, Ioannis; LAI, Matthew; GUEZ, Arthur; LANCTOT, Marc; SIFRE, Laurent; KUMARAN, Dharshan; GRAEPEL, Thore; LILLICRAP, Timothy; SIMONYAN, Karen; HASSABIS, Demis, 2017. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. arXiv. Available from DOI: 10.48550/ARXIV.1712.01815.

SOLOMONOFF, R.J., 1964. A formal theory of inductive inference. Part II. *Information and Control*. Vol. 7, no. 2, pp. 224–254. ISSN 0019-9958. Available from DOI: https://doi.org/10.1016/S0019-9958(64)90131-7.

SUTTON, Richard; MCALLESTER, David; SINGH, Satinder; MANSOUR, Yishay, 2000. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Adv. Neural Inf. Process. Syst.* Vol. 12.

SUTTON, Richard S.; BARTO, Andrew G., 2018. *Reinforcement Learning: An Introduction*. Second. MIT Press, Cambridge, MA. ISBN 0262039249.

TREISTER-GOREN, Anat; DUNIETZ, J; HUTCHENS, JL, 2000. The developmental approach to evaluating artificial intelligence–a proposal. *Performance metrics for intelligence systems*.

TURING, Alan M; HAUGELAND, J, 1950. Computing machinery and intelligence. *The Turing Test: Verbal Behavior as the Hallmark of Intelligence*, pp. 29–56.

TURING, Alan M., 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. Vol. 42, no. 1, pp. 230–265.

VADINSKÝ, Ondřej, 2018a. Towards General Evaluation of Intelligent Systems: Lessons Learned from Reproducing AIQ Test Results. *Journal of Artificial General Intelligence*. Vol. 9, no. 1, pp. 1–54. Available from DOI: doi:10.2478/jagi-2018-0001.

VADINSKÝ, Ondřej, 2018b. Towards General Evaluation of Intelligent Systems: Lessons Learned from Reproducing AIQ Test Results. *Journal of artificial general intelligence*. Vol. 9, no. 1, pp. 1–54. ISBN 1946-0163.

VADINSKÝ, Ondřej, 2019. An Overview of Approaches Evaluating Intelligence of Artificial Systems. *On data analysis and decision making*, pp. 201–213. Available also from: http://cjs.utia.cas.cz/proceedings.pdf.

VADINSKÝ, Ondřej, Unreleased. Comparing Proximal Policy Optimization and Deep Q-Learning using Algorithmic Intelligence Quotient Test. Unreleased.

VENESS, Joel; NG, Kee Siong; HUTTER, Marcus; SILVER, David, 2010. Reinforcement Learning via AIXI Approximation. *CoRR*. Vol. abs/1007.2049. Available from arXiv: 1007.2049.

WATKINS, Christopher, 1989. Learning From Delayed Rewards.

WEBER, Théophane; RACANIÈRE, Sébastien; REICHERT, David P.; BUESING, Lars; GUEZ, Arthur; REZENDE, Danilo Jimenez; BADIA, Adria Puigdomènech; VINYALS, Oriol; HEESS, Nicolas; LI, Yujia; PASCANU, Razvan; BATTAGLIA, Peter; HASSABIS, Demis; SILVER, David; WIERSTRA, Daan, 2017. *Imagination-Augmented Agents for Deep Reinforcement Learning*. arXiv. Available from DOI: 10.48550/ARXIV.1707.06203.

WOLPERT, David H.; MACREADY, William G., 1997. No free lunch theorems for optimization. *IEEE Trans. Evol. Comput.* Vol. 1, pp. 67–82.

WU, Yuhuai; MANSIMOV, Elman; LIAO, Shun; GROSSE, Roger; BA, Jimmy, 2017. *Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation*. arXiv. Available from DOI: 10.48550/ARXIV.1708.05144.

Appendices

A. Folder structure

This part of the appendix will describe the structure of additional files provided along with the text.

- **scripts** - Folder for all assistive scripts
  - **batch_script_template.sh** - Template script for creation of batch processes.
  - **qsub_generator_template(PPO).sh** - Template script for grid search of parameters.
- **AIQ-2023_PPO-VPG_def.csv** - CSV file with results acquired by running AIQ test over various configurations of parameter Steps_per_Epoch.
- **zemp02_DP.pdf** - A copy of the text of the thesis.

The contents of the "AIQ-2023_PPO-VPG_def.csv" file headers are based on triplets of agent results of the format: agent_name(SPE configuration value) denoting from what configuration the results of that column are, with suffixes "+-" describing column with margin of error of that configuration and "SD" standard deviation of that configuration.

The final state of the AIQ test code based on Python 3, containing new improvements and agents can be found in the `https://github.com/zemp02/AIQ/tree/AIQ-2023-Thesis` branch and repository.

B. Metacentrum Scripts User Guide

This appendix will contain guides explaining how to work with templates for generating combinations of parameters for agents and creating a batch processing request for each of the generated combinations. There are two scripts that together allow for the mass creation of grid search batch processing requests. A censored template will be provided along with steps defining how to fill this template.

B.1 batch_script_template

This script is based on a script originally provided by Ing. Vadinsky, PhD. It specifies steps that will be taken by every batch processing request to prepare parameters, set up the environment, run the experiment and export data.

1. Replace text *#ReplaceWithPathToHome* with the full path to the top folder of your experiment (Data Directory).
2. Replace text *#PathToAIQFromDatadir* with the path to your AIQ folder from Data Directory.
3. Replace text *#PathToYourAIQEnvironment* with the path to your conda environment for AIQ.
4. Set up parameters for AIQ inside their lists.
   - machines - reference machines to be used in AIQ.
   - episodes - episode length to be used in AIQ.
   - samples - number of samples to be used in AIQ.

B.2 qsub_generator_template(PPO)

The second script generates commands based on lists defining each value to be tested for each parameter. Provided template is based on the PPO agent and allows for the dynamic generation of configurations that are then passed to the qsub command template script. The provided template contains ten parameters defined in the list of parameters. Due to the code used, it is required that provided parameters are in backward order to the way they are normally provided to the agent. To prepare the template, the following must be done:

1. Define the agent to be tested in list agents.
2. Create a list of possible values for each of the defined parameters
3. Define all list names for parameters of the agent in backward order in list parameters with the list for the agent on the bottommost row.
4. Configure -N and -l parameters of the qsub command on line 93 as you need.
5. Start the generation of qsub commands with "*bash Path to qsub_generator script*"

The main part of the code generating string with combinations of agent parameters looks like this:

```
# Generate the command
cmd="printf ',%s\n' \${${parameters[0]}[@]} | "
for i in "$\{parameters[@]:1:${#parameters[@]}-2\}";  do
    cmd+="xargs -I{} printf ',%s{}\n' \${${i}[@]} | "
done
cmd+="xargs -I{} printf '%s{}\n' \${${parameters[-1]}[@]}"

# Execute the command and save each line as a separate value in an array
readarray -t array <<< "$(eval "$cmd")"
```

The generated string is then passed to a batch processing request as extra parameter AGENTS. Each batch processing request utilises a generic configuration template file defined in section B.1. This configuration file defines the entire process during batch processing. This file also accepts the AGENT value as a separate parameter passed to the initialisation of the AIQ process.