An Approximation of the Universal Intelligence Measure on Selected Environments

Bachelor Thesis

by

Christian Hauser

Degree Course: Computer Science

Matriculation Number: 1933900

Institute of Applied Informatics and Formal Description Methods (AIFB)

KIT Department of Economics and Management

Advisor:	*Titel Vor- und Nachname*
Second Advisor:	*Titel Vor- und Nachname*
Supervisor:	M.Sc. Konstantin Pandl
Submitted:	July 9, 2020

Abstract

This thesis covers an approximation of the Universal Intelligence Measure by Legg and Hutter [5]. The goal is to test an agent's ability to achieve rewards in a wide range of environments. Python is used as a Turing Machine, Kolmogorov Complexity is approximated with python bytecode instructions and environments are handpicked. We then test our measure called Approximated Python Intelligence Quotient several statistics based agents and show that they are ordered sensibly. We also evaluate some neural network based agents, which are also ordered as expected and reveal interesting differences between the sigmoid and the relu activation function.

Contents

List of Abbreviations

AGI Artificial General Intelligence.

AIQ Algorithmic Intelligence Quotient.

APIQ Approximated Python Intelligence Quotient.

IQ Intelligence Quotient.

UIM Universal Intelligence Measure.

1 Introduction

Artificial General Intelligence (AGI) is in many aspects like the hydrogen fusion reactor: Very sought after, highly dangerous if done wrong, and a long time in the making, with significant progress, yet many unsolved problems. With the resurgence of deep learning, mostly by the merit of increasing computational resources, systems capable of human or even superhuman performance have been constructed for many highly complex domains. Notable examples include image classification [3] and playing Go [7] and Starcraft 2 [9]. Although those achievements are highly impressive in their own right, none of them qualify as AGI. They are highly specialized and incapable of handling tasks other than what they are designed for without massive human intervention.

AGI has no broadly accepted definitions but many key likely properties can be agreed upon [1]:

- Ability to achieve a variety of tasks in a variety of environments.
- Ability to handle problems not anticipated by its creators.
- Ability to generalize knowledge and transfer it from one context to another.
- Better generalizing ability in some context than in others, given limited resources.
- Exhibition of some or all of the following traits: Pattern recognition, classification, reasoning, induction, deduction, creativity, association, generalization, problem-solving, memorization, planning, learning, and optimization.

One of the main reasons, why AGI has no broadly accepted definition is that intelligence itself is defined in many different ways. In their 2007 paper [4], Legg and Hutter survey about 70 different definitions and conclude that intelligence:

- Is a property that an individual agent has as it interacts with its environment or environments.
- Is related to the agent's ability to succeed or profit concerning some goal or objective.
- Depends on how able the agent is to adapt to different objectives and environments.

From those traits, they adopt the definition:

> "*Intelligence measures an agent's ability to achieve goals in a wide range of environments.*"

This is then formalized in their 2007 paper [5] as the Universal Intelligence Measure (UIM). This measure is illustrated in the theory part of this thesis. It can roughly be described as the ability of an agent to maximize the received reward in every computable environment.

1.1 Problem

The UIM is not computable for several reasons.

- Kolmogorov complexity is applied as a measure for the complexity of the environments. For a binary string s it is defined as the shortest program that computes s and it is not computable for non-trivial cases. An upper bound can be given by providing an example program that computes s.
- The class of all computable environments is infinite and can therefore only be sampled from.
- The reward an environment can return is bounded by 1, but the number of steps it takes to receive a significant reward from an environment can be infinite. In practical implementations, this necessitates a maximum number of steps an environment is evaluated for.

In [6] Legg and Veness approximate the UIM. They use a subset of C commands as their Turing Machine where every possible program is valid and randomly sample programs from it. This makes estimating Kolmogorov complexity simple.

A drawback of sampling randomly is that we cant choose interesting environments to evaluate. If we choose them by hand, it biases our sample towards what we consider interesting or meaningful but also gives us the chance to test agents in handpicked environments for solving which we would consider an agent intelligent.

1.2 Goal

The goal of this work is to create a framework for approximating the UIM with handpicked environments. This framework should have the following traits:

- Approximate the UIM for implemented agents. The expectation is that the results for the agents are positively correlated with their ability to "achieve goals in a wide range of environments".
- Approximate Kolmogorov complexity of environments.
- Allow easy addition of agents to be evaluated.
- Allow easy addition of diverse environments to increase the accuracy and significance of results.
- Save computed results so recomputation is minimal as new agents and environments are added.

- Computable in a reasonable amount of time with a consumer pc. This places complexity constrains on environments and agents. It also limits the number of evaluations and therefore the accuracy of results in the presence of randomness.
- The resulting approximated UIM is a normed value between 0 and 1. An agent taking random actions should receive 0. The maximum possible value is 1. Since environments can have random elements, the practical maximum possible value is somewhere between 0 and 1.

2 Theory

2.1 Machine Learning

In Machine Learning, the desire is for artificial agents to learn to transform varying inputs into desired outputs. There are many reasons this is desirable and even more methods for achieving it.

Machine Learning is especially useful when an agent can find better solutions for problem domains than humans. One example of this is chess, where humans cannot hope to compete any longer since Kasparov lost to Deep Blue in 1997, and where recently machine learning methods have arguably overtaken even the brute force, expert rule tuned Stockfish [8].

Another important application is replacing the necessity for human intervention. Consider spam mails where trained humans are capable of unmasking all but the most intricately designed ones, but don't have the capacity or resources to check every mail. It is hard to hardcode rules for detecting spam but machine learning methods have made it possible to learn policies which have reduced the number of spam emails that pass filters by staggering amounts.

Machine learning can roughly be split into three categories: Supervised learning, unsupervised learning, and reinforcement learning.

Supervised Learning is when the input data is labeled with the correct output, and the learning method is corrected according to the difference between the label and its output. In neural networks, this is done with the backpropagation algorithm.

Unsupervised Learning is primarily used for finding structure in data. There are a multitude of algorithms like k-nearest neighbors and principal component analysis to explore the structure of data without the need for a human to label it first.

Reinforcement Learning is an extremely general framework used for many different machine learning applications. It is also used for the UIM and will, therefore, be explored in more detail. The basic idea is illustrated in figure 1. An agent sends an action to an environment and the environment, in turn, sends an observation and a reward signal to the agent. The example for intuition here is a human child learning that certain foods are yummy and that putting their hand on the stove hurts. There is no ideal action to compare to, but by punishment and reward, certain actions are discouraged or incentivized.

The advantage of this method is, that an agent can be nudged closer to an abstract goal like survival in the case of the child without explicit knowledge of how to get there.

It is, however, unlikely to find an ideal solution in a complex environment with just

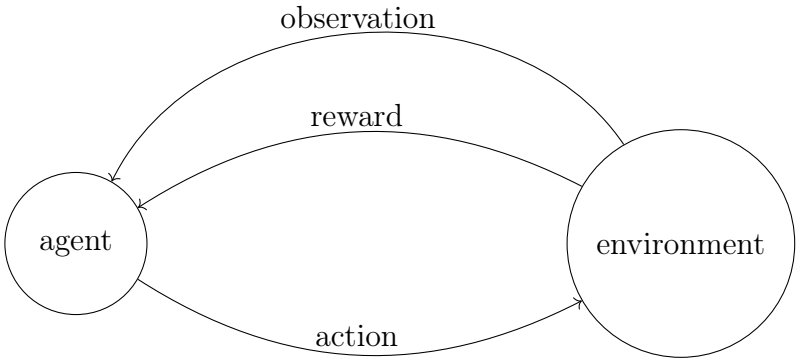

Figure 1: Reinforcement learning framework

reinforcement learning. The process tends to get stuck in local minima, where every change in strategy decreases the reward signal. This is akin to a child having to learn that sometimes enduring pain yields better longterm results, like when doing boring homework is required for better grades.

2.2 Universal Intelligence Measure

2.2.1 Derivation

Notice how the reinforcement learning framework in figure 1 lends itself very well to the informal definition of intelligence from the introduction by Legg and Hutter [4]:

> "*Intelligence measures an agent's ability to achieve goals in a wide range of environments.*"

Agent and environment are present and the goals are represented by the reward signals sent by the environment.

Now to formalize the definition we follow the path laid out in [5] and start with the signals between agent and environment. The actions a sent from the agents are elements of the *action space* $\mathcal{A}$, which is a finite set of symbols. Similarly, the environment sends symbols p from the *perception space* $\mathcal{P}$. Every perception consists of an observation o and a reward r. o is taken from the *observation space* $\mathcal{O}$ and r is from the *reward space* $\mathcal{R}$, which is a subset of $[0, 1] \cap \mathbb{Q}$.

Agent. The agent is a function π, which takes the current history as input and chooses the next action as output. This function does not have to be deterministic. It is conveniently represented as a probability measure for actions depending on the interaction history. For example, $\pi(a_2|o_1r_1a_1o_2r_2)$ is the probability of action a_2 given the history $o_1r_1a_1o_2r_2$.

This definition is extremely general and leaves π open to be anything from a human to a function computing the digits of e to an artificial agent taking completely random actions.

Symbols a, o, and r will be indexed in the order in which they occur. Agent and environment take turns sending symbols to each other, starting with the environment. This produces a history $o_1r_1a_1o_2r_2a_2o_3r_3a_3o_4...$.

Environment. The environment is defined similarly as a function μ. The probability for observation and reward o_kr_k at step $k \in \mathbb{N}$ given the interaction history $h = o_1r_1a_1o_2...o_{k-1}r_{k-1}a_{k-1}$, is given by the probability measure $\mu(o_kr_k|h)$.

The Measure of success. Success for an agent π is measured by how much reward it can receive from the environment μ. For some environments, it might be best to greedily take as much reward as possible as soon as a strategy seems to be working. For others, first optimizing the strategy will bring greater longterm rewards. The only condition on the environment is, that the total reward returned can never exceed 1. This also means that the expected value of the sum of rewards V_μ^π is smaller or equal to 1, as described in equation 2.1.

$$V_\mu^\pi := \mathbf{E}\left(\sum_{i=1}^{\infty} r_i\right) \leq 1 \tag{2.1}$$

Space of environments. The informal definition talks about a *range of environments* and intelligence is in part about learning about and adapting to different problems and environments, so we want our space of environments to be as big as possible. Well suited for this is the set of computable probability measures. It is infinite, places no limits on the complexity, and even allows non-deterministic environments since for example, a probability measure describing an infinite sequence of random coinflips is computable. Legg and Hutter assume this to be the largest reasonable space of environments [5, p.20].

Occam's razor. When faced with the question of what the next number is in the sequence $2, 4, 6, 8$, most would immediately recognize a simple pattern and answer 10. However, the polynomial $2k^4 - 20k^3 + 70k^2 - 98k + 48$ is also consistent with the data and produces 58 as the next number. The fact that 10 is considered the most likely and is also defined as the "correct" answer in intelligence tests is rooted in our application of Occam's razor:

"*Given multiple hypotheses that are consistent with the data, the simplest should be preferred.*" [5, p.21],

Agents don't know about the probability measure describing the environment and can only make hypotheses about it from the interaction history. Since there will almost always be many applicable hypotheses, we want agents to invoke Occam's razor to reason about which hypotheses are likely to be correct. Inferring from this we want to reward them for correctly applying Occam's razor and thus for doing good in less complex environments. We, therefore, add a scaling factor to the reward accumulated in an environment depending on the complexity of the environment.

Complexity. To measure the complexity of environments we use Kolmogorov complexity of a binary string x. It is defined as the length of the shortest program that computes x as described in equation 2.2.

$$K(x) := \min_p \{l(p) : \mathcal{U}(p) = x\} \tag{2.2}$$

p is a binary string which we call a program, $l(p)$ is the length of this string in bits and $\mathcal{U}$ is a prefix universal Turing machine called the *reference machine*. Intuitively, a string that can easily be computed will have a low Kolmogorov Complexity. For example, a string of a trillion 0s can be computed by a loop adding a 0 a trillion times and thus has low complexity. In contrast, a long irregular random string is more difficult to find a short program for. It can even be shown, that there are so many complex strings of this kind relative to the number of short programs, that most long random strings cannot have short programs computing them. This means most long random strings have high Kolmogorov complexity.

An important property of K is that it is independent of the choice of the *reference machine* $\mathcal{U}$ exempt a finite additional cost for a different machine $\mathcal{U}'$ to simulate $\mathcal{U}$ which will be small for reasonable Turing machines.

Applied to the probability measure for an environment, its complexity will be low if there is a short program that describes it and vice-versa.

Scaling factor. Now that we can measure the complexity of our environments, we need to formalize Occam's razor so simpler environments are valued more. Considering that each environment is described by a minimal length program that is a binary string, a natural way to do this is to multiply the scaling factor with one half for each additional bit, reflecting the two choices for every bit. This gives us the *algorithmic probability distribution* over the space of environments, $2^{-K(\mu)}$.

Universal Intelligence. The Universal Intelligence Measure Υ for an agent π can now be defined as:

$$\Upsilon(\pi) := \sum_{\mu \in E} 2^{-K(\mu)} V_\mu^\pi. \tag{2.3}$$

Here μ is an environment from the space E of all computable reward summable probability measures concerning the reference machine $\mathcal{U}$, K is the Kolmogorov complexity and V_μ^π is our expected reward from equation 2.1.

2.2.2 Review

The formal definition captures all the important parts of the informal one, all while being extremely general. There is the agent π being evaluated in all reward summable environments μ from the set E of computable measures. The reward is implicit in the environment and the agents' performance is represented by V_μ^π. Occam's razor is given by $2^{-K(\mu)}$, which weights the agent's performance in each environment inversely proportional to its complexity. Finally, the communication channels are very general and there are no restrictions on the internal workings of agents, which makes UIM extremely general.

2.2.3 Evaluation of agents

In its current form, the UIM is not computable since Kolmogorov complexity is not computable. For this reason, this work aims to create a workable test that approximates an agents' Υ value. It is, however, possible to order some agents with general properties after their universal intelligence to gain an intuition about its workings.

A random agent. This agent will be called π^{rand}. It is the agent with the lowest Υ among those who aren't trying to perform badly on purpose. It will fail to take advantage of any regularities in the environment, no matter how simple. $V_\mu^{\pi^{rand}}$ won't always be low though since some environments will generate a high reward for every action.

A very specialized agent. This agent will be called π^{dblue}. Like the chess computer deep blue does extremely well in a very narrow domain but extremely bad in others. This means its universal intelligence will be low since Υ will only be high if the agent does well in many environments and even more so if the narrow domain is highly complex since it gets more heavily discounted by Occam's razor.

A general but simple agent. This agent will be called π^{basic}. It performs basic learning with a table of action and observation pairs and statistics about which rewards follow from which actions. 90% of the time it will choose the action with the highest expected

reward and 10% of the time a random action. Many environments will have at least some structure this agent can take advantage of. This means for most μ: $V_\mu^{\pi^{basic}} > V_\mu^{\pi^{rand}}$ and therefore $\Upsilon(\pi^{basic}) > \Upsilon(\pi^{rand})$. This lines up with what we would expect from a measure of intelligence, since π^{basic} is surely more intelligent than π^{rand}.

In a similar vein π^{dblue} won't take advantage of the simplest regularities in most environments. Thus we conclude $\Upsilon(\pi^{basic}) > \Upsilon(\pi^{dblue})$. This also matches our expectations, since an agent that can exploit simple regularities should have higher intelligence than one which only does well in a very narrow domain.

A simple agent with more history. This agent will be called π^{2back}. This agent will use a similar structure to π^{basic} but it will be able to take into account 2 cycles of action and perception when choosing the action with the highest expected reward. This will give it the ability to for example exploit an environment that gives reward only if the next action is different from the last, something which π^{basic} cannot do. We conclude, that in general $V_\mu^{\pi^{2back}} > V_\mu^{\pi^{basic}}$ and so $\Upsilon(\pi^{2back}) > \Upsilon(\pi^{basic})$. This lines up with our intuition since an agent that takes the history into account with otherwise the same capabilities should be more intelligent than an agent which doesn't. Also the further we increase the length of the history taken into account, the higher the intelligence should get.

A simple forward-looking agent. This agent will be called $\pi^{2forward}$. Similarly to taking the history into account, if the agent is additionally able to plan its actions can take advantage of more regularities and thus have higher intelligence. As an example, picture a slide, where staying at the bottom gives a low reward, going up gives no reward, and sliding down gives a high reward. π^{basic} and $\pi^{2backward}$ will stay at the bottom of the slide because of the higher expected reward. $\pi^{2forward}$ will try to maximize the average reward for the next 2 actions and take advantage of the high reward from using the slide the intended way. What follows is, of course, $\Upsilon(\pi^{2forward}) > \Upsilon(\pi^{2backward})$ and by extension all aforementioned agents as we would expect for the intuitively more powerful $\pi^{2forward}$.

A very intelligent agent. An agent with a high Υ would have to perform well in simple environments and reasonably well in more complex environments and vice versa performing well in simple environments and reasonably well in complex environments would ensure a high Υ. Such an agent would have to take into account the complete history and find regularities in it, as well as plan future actions up to a reasonable point and maximize the expected reward.

A super-intelligent agent. This agent is called π^{AIXI}. By definition, a "perfect" agent would always pick the action which has the greatest expected future reward. For this it

would have to consider for every environment $\mu \in E$ how likely it is facing this environment given the interaction history and the prior probability of μ which is $2^{-K(\mu)}$. Then it would consider all possible future interactions that might occur, how likely they are and from this select the action that maximizes the expected future reward. Such an agent would have maximal Υ.

This perfect theoretical agent is named AIXI and is defined and studied in [2]. With it, the upper bound to UIM $\bar{\Upsilon}$ can be defined as in equation 2.4.

$$\bar{\Upsilon} := \max_{\pi} \Upsilon(\pi) = \Upsilon(\pi^{AIXI}). \tag{2.4}$$

AIXI is not computable since K isn't computable, but it is interesting from a theoretical standpoint. It has been proven, that AIXI converges to optimal performance where this is at all possible for a general agent, which proves that agents with very high universal intelligence are extremely powerful and general.

A human. A human should be able to find simple regularities in environments and depending on training, tools, and time for analysis even more complex ones. It is, however, difficult to estimate the universal intelligence of a human since much of the human brain is rather specialized for processing information from sense organs and managing the body.

2.3 An Approximation of the Universal Intelligence Measure

Given that UIM is not computable, to construct a practical test for intelligence we have to make approximations and design choices. In this section, we will go through the choices made in Legg and Veness "An Approximation of the Universal Intelligence Measure" [6] so we can later contrast them to the choices made in this thesis. They call their measure Algorithmic Intelligence Quotient (AIQ).

Sampling environments. Environments are sampled by consecutively adding random bits until a program stop symbol is reached. The chance for program p is then given by $2^{-l(p)}$. The length of program p, $l(p)$, is consequently an upper bound to the Kolmogorov complexity of the string s that p computes. Since the chance of sampling a program already weights environments like our formal definition of Occam's razor, no further weighting factor is required. The changed equation 2.3 can be seen in equation 2.5. Notice that the expectation V_{μ}^{π} is replaced with $\hat{V}_{p_i}^{\pi}$, which is the empirical total reward returned from a single trial of environment $\mathcal{U}(p_i)$ interacting with agent π.

Symbol	Description	C
>	move pointer right	p++;
<	move pointer left	p−−;
+	increment cell	∗p++;
−	decrement cell	∗p−−;
.	write output	putchar(∗p);
,	read input	∗p = getchar();
[	if cell is non-zero, start loop	while(∗p) {
]	return to start of loop	}
%	write random symbol to cell	∗p = rand();

Table 1: Commands of Reference Machine for AIQ

$$\hat{\Upsilon}(\pi) := \frac{1}{N} \sum_{i=1}^{N} \hat{V}_{p_i}^{\pi} \tag{2.5}$$

Environment simulation. Programs that do not halt within the allotted computation time cannot be evaluated. Therefore any programs this is the case for, will be discarded. This limit can be increased as more powerful hardware becomes available.

Temporal preference. Since the total reward that an environment can return is upper bounded by one, the question of reward allocation becomes important. We have no way of knowing whether a given program will respect this bound, so further constrictions are necessary. One elegant solution would be to allow the agent to return a bounded reward in every cycle and use geometric discounting. We can then terminate an evaluation once the possible remaining reward drops below a certain value. This would mean, however, that the later cycles where the agent has learned the most are the most heavily discounted. The alternative used in AIQ then, are undiscounted bounded rewards over fixed length trials.

Reference machine selection. The BF language by Urban Müller was used as a reference machine and slightly altered to fit the reinforcement learning paradigm. It consists of an input tape, an output tape, a work tape and 8 symbols which are a subset of C commands. A 9th symbol % was added, which writes a random symbol to the current work tape cell and allows for non-deterministic environments. Table 1 shows a list of all symbols and their corresponding commands. The first bit in a program determines if the returned reward will be negated. Since this bit is set randomly, it causes randomly acting agents to have an AIQ of zero.

3 Approximated Python Intelligence Quotient

The name Approximated Python Intelligence Quotient (APIQ) was chosen for very literal reasons. Firstly it is necessarily an approximation of the UIM since the UIM isn't computable. Secondly, the Python programming language is used as a reference Turing Machine. Thirdly the Intelligence Quotient (IQ) is generally understood to be a measure of intelligence, although it isn't even a quotient anymore, but based on standard deviation.

This section has two parts. The first part focuses on the approximations made in APIQ and their advantages and disadvantages and contrast is drawn to Legg and Veness work [6]. The second part details the architecture decisions made.

3.1 Approximations

Reference Turing Machine. The Turing Machine chosen for this approximation of the AIQ is the python programming language. It is, of course, more complex than the small subset of C commands used in [6]. This means estimating the complexity of environments is more difficult than just counting the bits of a program. It also means random sampling isn't as easy since not every program is valid. To our advantage, our agents can natively use the rich machine learning ecosystem that python provides.

Python lends itself very well to our more opinionated measure of intelligence. To measure intelligence comparable to humans, a Turing Machine that more closely resembles our surrounding world might be required since the chosen Turing Machine has a significant effect on the measured intelligence of all but the most intelligent agents. Such a Turing Machine would, however, be highly complex and possibly require much more computation resources than we have available.

Kolmogorov Complexity. Since Kolmogorov Complexity is not computable, we approximate its upper bound by providing a program that computes the environment. We then count the number of python bytecode instructions with the disassembler. Every instruction has a length of 2 bytes, so we could give the complexity in bits, but since there is no smaller increment between two programs with different complexity than 2 bytes, the complexity is given in bytecode instructions. In contrast, Shane and Legg's approach was to use Monte Carlo sampling of environments where the chance for a specific environment already weights that environment, which is why there is no need for an additional scaling factor.

Our measure of complexity will be denoted $\hat{K}_{\mathrm{P}}$ to show its derivation from Kolmogorov Complexity and its relation to Python. It is formulated in equation 3.1 following equation 2.2.

$$\hat{K}_{\mathrm{P}}(x) := \min_{found}\{l(p) : \mathrm{P}(p) = x\} \tag{3.1}$$

Sampling Environments. The manual choice of environments in this work is probably the greatest difference to [6]. Possible environments are evaluated on their ability to test agents on the following dimensions to assess whether they make a valuable addition to the pool of environments:

- Handling variety.
- Handling complexity.
- Taking the history into account.
- Planning for future rewards.
- Context switching.
- Generalization.

The implemented environments can be found in Appendix A.1 and a selection is used for further illustration in section 3.2.

The manual choice of environments lets us efficiently test agents for desirable properties in the context of intelligence. Consequently, it introduces a bias for the kind of environments we choose. The agents still get evaluated in a wide range of environments though, so we expect the resulting measure to still be valid.

Reward. In the UIM the only constrain on rewards in environments is that their sum can't exceed 1. One way to achieve this is by using geometric or harmonic discounting of rewards. This, however, leads to the later turns where the agents have learned the most being discounted the most. In this thesis, we instead follow the lead of [6] and use fixed bounded rewards. There is a fixed amount of cycles and in every cycle, the environment can return a maximum reward of 1. The sum of rewards is then divided by the number of cycles. The reward is further divided by the maximum average reward per cycle for the given environment, so environments that require multiple cycles of setup for a reward don't get discounted. This is formalized in equation 3.2 following equation 2.1 with the empirical reward $\hat{V}_{\mu}^{\pi}$, agent π, environment μ, number of cycles n, reward in cycle i, r_i, and maximum average reward $\tilde{r}$.

$$\hat{V}_{\mu}^{\pi} := \frac{1}{n \cdot \tilde{r}} \cdot \sum_{i=1}^{n} r_i \tag{3.2}$$

Occam's Razor. If our programs that simulate environments are bitstrings, the number of different programs of length n is 2^n. Accordingly in UIM the *algorithmic prior probability* 2^{-n} is used to weight the reward accumulated in the environment simulated by a program of length n. Since we don't test all programs, but a finite biased sample, we need to adjust the weighting. If we would test environments so that every complexity appears exactly once, there would be no need for a weighting since every environment is a sample of the 2^n environments of the same complexity, weighted by 2^{-n}. This is, however, whether feasible nor desirable in our case, since we are interested in environments that test the agents' abilities and don't want to satisfy some arbitrary complexity constrains.

For this reason, we weight environments based on how many environments with similar complexity there are, relative to the total number of environments and the range of complexity for which we have implemented environments. As a basis for this, we take a normal distribution $\mathcal{N}_{0,\sigma}$, shown in equation 3.3 and the discrete distribution of complexity $\rho_d(x)$, shown in equation 3.4. $\mathcal{E}_+$ is the set of implemented environments and σ^2 is the average density of environments in the range from the most simple environment to the most complex environment.

$$\mathcal{N}_{0,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{x^2}{2\sigma^2}),\ \sigma^2 = \frac{\max_{\hat{K}_\mathrm{P}(\mathcal{E}_+)} - \min_{\hat{K}_\mathrm{P}(\mathcal{E}_+)}}{|\mathcal{E}_+|} \tag{3.3}$$

$$\rho_d(x) := |\mu : \hat{K}_\mathrm{P}(\mu) = x| \tag{3.4}$$

We then convolve these functions to get a continuous function $\rho_c(x)$ which takes the relative density of environments with similar complexity into account. The resulting weighting $\omega(x)$ is formalized in equation 3.5. It has the desirable property that implementing more environments within the existing range of complexity decreases σ and therefore $|\mathcal{E}_+| \to \infty \Rightarrow \rho_c \to \rho_d$.

$$\omega(x) := \frac{1}{\rho_c(x)} := \frac{1}{(\rho_d \circ \mathcal{N}_{0,\sigma})(x)} \tag{3.5}$$

In figure 2 we can see how the continuous density distribution takes the neighboring complexities into account, while the discrete density distribution does not.

APIQ. Since we want our measure to be normed between 0 and 1 we have two more adjustments to make. Firstly, we divide the result by the sum of the weights $\sum_{\mathcal{E}}(\omega(\hat{K}_\mathrm{P}(\mu)))$. This sets the maximum achievable APIQ to 1. Secondly, we double the pool of environments by including a clone of every environment where the reward is negated. This makes a randomly acting agent, the agent who should do worst among the agents who aren't trying to do bad on purpose, receive an APIQ of 0. The environments with negated reward have

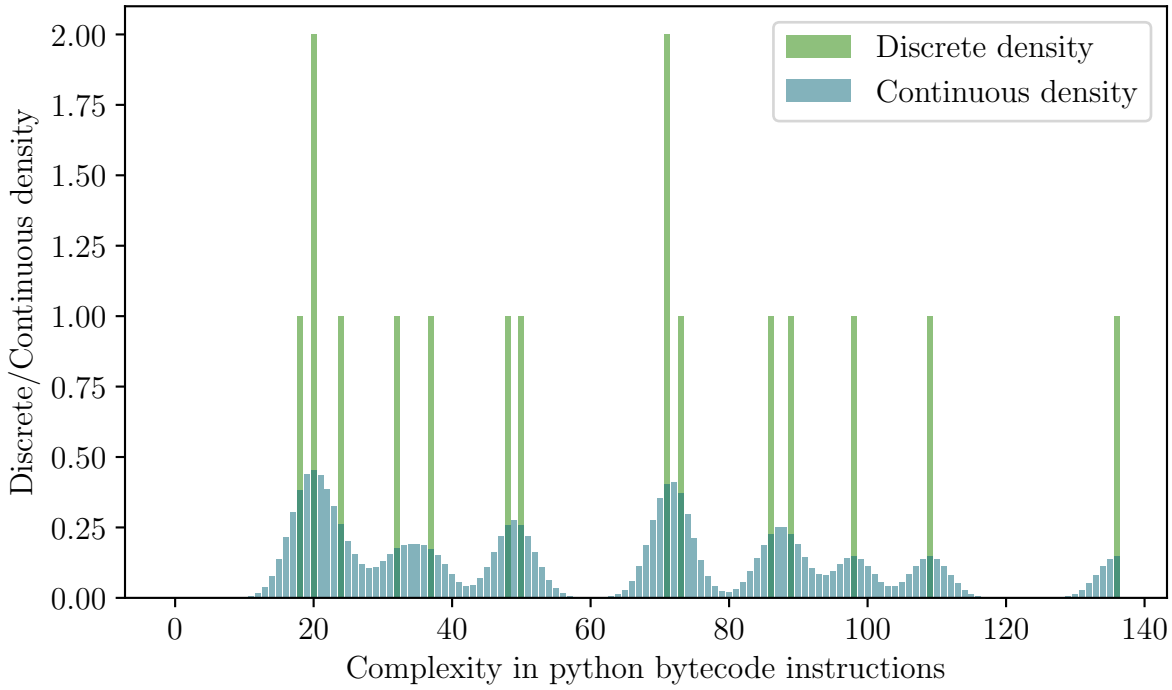

Figure 2: Comparison between the discrete and the continuous distribution of complexity of environments

a maximum acummulated reward of 0. $\mathcal{E}$ is the set of all implemented environments and their negations. Combining all the parts results in equation 3.6 for the APIQ of agent π, $\hat{\Upsilon}_{\mathrm{P}}(\pi)$. The weighted rewards are summed and then divided by the sum of weights. The fraction is doubled because the maximum of $\hat{V}_{\mu}^{\pi}$ is 1 for the implemented half of environments and 0 for the negated half.

$$\hat{\Upsilon}_{\mathrm{P}}(\pi) := 2 \cdot \frac{\sum_{\mathcal{E}}(\hat{V}_{\mu}^{\pi} \cdot \omega(\hat{K}_{\mathrm{P}}(\mu)))}{\sum_{\mathcal{E}} \omega(\hat{K}_{\mathrm{P}}(\mu))} \tag{3.6}$$

3.2 Architecture

3.2.1 Signals

The signals between environments and agents are actions and percepts. Actions are represented as bitstrings and percepts as tuples of two bitstrings which represent observation and reward. The minimum length for action, observation and reward strings is 1, 0 and 2 respectively. The first bit of the reward string determines its sign and every further bit adds $(1/2)^{n-1}$ to the reward depending on its index n in the string. This means for example, that the reward string "101" translates to a reward of -0.5. Environments must ensure that rewards never exceed |1|.

3.2.2 Environments

Environments inherit from the abstract parent class `environment.py` and implement the abstract method `calculate_percept`, which takes an action as parameter and returns a percept. They also specify up to 5 parameters if they vary from their defaults in `Environment.py`. Those parameters and their defaults are:

- `observation_length`: 0
- `reward_length`: 2
- `action_length`: 1
- `max_average_reward_per_cycle`: 1
- `has_randomness`: False

The length parameters are used to signal the length of the bitstrings to the agents. The `max_average_reward_per` cycle signals how much reward an agent can obtain on average with optimal actions.

The environment has access to randomness by using `random.randint(0, 1)`, which randomly returns 0 or 1. The amount of python bytecode instructions for generating a pseudorandom 0 or 1 has been determined to be at most 25, so if the environment uses randomness, the parameter `has_randomness` is set to True and 25 will be added to the complexity of the environment when it is calculated.

The complexity of an environment is determined by counting the bytecode instructions of the `calculate_percept` method, so any use of imported methods except `random.randint(0, 1)` is forbidden.

A total of 16 environments have been implemented. The distribution of their complexity is the discrete density in figure 2. The following environments are descriptive examples of the kind of considerations that went into choosing environments to implement. A full list of environments can be found in Appendix A.1.

Button. A very simple environment with a $\hat{K}_{\mathrm{P}}$ of 18, is the button environment. It returns a reward of 1 if the action string is "1", otherwise it returns 0. Any agent able to exploit regularities in action-reward pairs will be able to get high $\hat{V}_{\mu}^{\pi}$ in this environment.

Alternate. A slightly more complex environment with $\hat{K}_{\mathrm{P}} = 24$ is the alternate environment. In even cycles it returns 1 if the action string is "1" and in odd cycles, it returns 1 if the action string is "0". This means to receive high $\hat{V}_{\mu}^{\pi}$ in this environment, an agent has to take into account its last action when choosing the next one.

Slide. The slide environment with a complexity of 37, tests the agents' ability to plan. The agent starts at the bottom of the slide and receives a reward of 0.125 for staying at the bottom by sending action "0" and a reward of 0 for climbing the slide with action "1". If the agent is at the top of the slide, it receives a reward of 1 and slides back down, no matter the action. Shortsighted agents will stay at the bottom for an average reward of 0.125, while an agent with some planning will climb up and slide down for an average reward of 0.5.

AlternateRandomly. This environment sits at $\hat{K}_{\mathrm{P}}$ 86 and was chosen to test an agents' ability to switch contexts. It has two states. It can return 1 if the action is "1" or 1 if the action is "0". It switches between those states at random with a chance of 6.25%. Agents who just collect simple action reward statistics won't be able to exploit that the chance of staying in a state is higher than switching states and so will receive little reward.

BiasedCoinFlip. This environment models a biased coin flip with a chance of 75% for heads and 25% for tails. Its complexity is 73. If the agent predicts the correct result, it gets rewarded. It tests the agents ability to deal with random events and choose the statistically best action.

3.2.3 Agents

Agents inherit from the abstract parent class `agent.py`, must implement the abstract method `calculate_action` and may implement the method `train`, which by default does nothing. They hold a reference to the current evaluated environment, where they extract the length of action, observation, and reward strings. The implemented agents are listed in A.2. Here we just provide a quick overview.

Pi Random. The simpler pi agents from the UIM theory section 2.2.2 have been implemented. The first is π^{rand}, which returns random actions. We expect it to have an APIQ of 0.

Pi Basic. π^{basic} chooses an action with simple observation, action, reward statistics. It has a dictionary with every possible observation as keys. The values to the keys are priority queues implemented as max heaps which contain tuples of `(expected_reward, action, cnt)` for every action. They contain the expected reward for their respective action and the number of times this action has been selected with the given observation. π^{basic} chooses a random action 10% of the time. The other times it chooses the first element of the priority queue. The expected reward is then updated with the received

reward according to the formula

$$r_{exp}^{new} = \frac{r_{exp}^{old} \cdot \texttt{cnt} + r}{\texttt{cnt} + 1}.$$

Then `cnt` is incremented by 1. We expect π^{basic} to have higher APIQ than π^{rand}.

Pi 2Back. π^{2back} is similar to π^{basic}. The only difference is, that statistics are kept for last_observation - action - observation sequences. The choice of action is still 10% random and 90% the max value of the corresponding priority queue and the updating still works with the same formula. Since there are more table entries, the entries will take longer to fill than in π^{basic}, so π^{2back} might be slower to learn certain regularities, but in return, it can take the last step into account and so choose a better action. We, therefore, expect π^{2back} to have higher APIQ than π^{basic}.

Pi 2Forward. $\pi^{2forward}$ has the same structure as π^{2back}, except the priority queue contains tuples with 2 actions and the expected reward for the next 2 rewards. This means, it is a more powerful agent than π^{2back} and we expect a higher APIQ. However in many environments where there are more complex regularities than taking the next reward or the last observation into account, $\pi^{2forward}$ will still do poorly, so if enough of those environments are implemented, we expect the APIQ of $\pi^{2forward}$ to still be rather low.

Handcrafted. Since the environments are implemented by hand and none of the environments yet are too complex for humans to solve, we can give a handcrafted solution to every environment. This agent will achieve an APIQ close to 1, only receiving rewards smaller than 1 in environments with randomness. It serves as a bound for all learning agents with no previous knowledge about the environments and we theorize its APIQ to be higher than the theoretical optimal agent AIXI which converges to the optimal solution.

NNAgents. Simple neural network based agents were implemented. In every cycle, they consecutively feed the observation together with every possible action to their neural network, whose output is the expected reward for this cycle. The action with the highest expected reward is chosen 90% of the time, and a random action 10% of the time. The activation functions, as well as layers and width of the neural network, are what defines the different neural network based agents. When the reward is received it is compared to the expected reward and the mean squared error function is used for backpropagation and adjustment of the weights and biases. We expect these agents to have an APIQ between π^{rand} and π^{basic} since they should be able to exploit simple regularities but take longer to do so than the table based version.

3.2.4 Evaluation

When an evaluation begins we first calculate the complexity of all environments. We then load previous results and check which agents have already been trialed in which environments. The agent-environment pairs which still need to be trialed are collected in a set and given to a process pool that executes the trials in parallel. The results are accumulated in a dictionary. The APIQ of each agent is then calculated from that dictionary. This means we only have to trial agent-environment pairs multiple times if either the agent or the environment change.

4 Results

Every agent was trialed 100 times for 10,000 cycles in the positive reward and in the negative reward version of all environments as seen in equation 3.2. From the rewards achieved in the 100 trials, we then calculate the arithmetic mean and the standard deviation. We then calculate the APIQ according to equation 3.6. For the errors, we used gaussian error propagation. The APIQ of agents is visualized in figure 3, the rewards of the pi agents and the handcrafted agent in figure 4, and the reward of the neural network based agents in figure 5. The environments in the reward figures are ordered from lowest to highest complexity. The following paragraphs are a listing of all agents and their APIQ, as well as a closer look in which environments they achieved which rewards and why.

Handcrafted (0.888 ± 0.001)**.** The handcrafted agent exemplifies the maximum reward an agent can receive in an environment and its negated version. Its mean APIQ is 0.888. The reason it isn't one, is either due to randomness, like in the coin flip environments or due to it being impossible to avoid some negative reward in the negated environment like in the slide environment.

DoubleCoinFlip is an environment where you bet on the number of heads when throwing 2 coins at random. Even with the optimal choice of 1 times heads with positive reward and 0 or 2 times heads with negative rewards, the accumulated rewards will approach $0.5 + (-0.25) = 0.25$ for $n \to \infty$.

In contrast, Slide is an environment without randomness, but the optimal choice in the negated reward version is to stay at the bottom of the slide and accumulate the negative reward of -0.25 instead of going up and sliding down for a negative reward of -1. This makes the maximum achievable reward in Slide_+ and Slide_- 0.75.

PiRand (0.000 ± 0.002)**.** The sum of the positive and negative rewards for trials in all environments is 0 or almost 0. Consequently, the mean APIQ of π^{random} is 0.000. This means the random agent behaves as expected, so we have a natural baseline for our intelligence measure.

PiBasic (0.117 ± 0.004)**.** The basic agent that does simple statistics on observation, action, reward tuples receives more or equal reward than π^{rand} in all environments except Slide and SlideTwo. It also tends to do better in environments of lower complexity. The reason for the negative reward in Slide and SlideTwo is that these environments will reward π^{basic} for staying at the bottom in the positive reward case and for climbing up in the negative reward case which incentivizes π^{basic} to take the worst possible actions. Since environments like Slide, that punish agents for shortsightedness are the exception and not

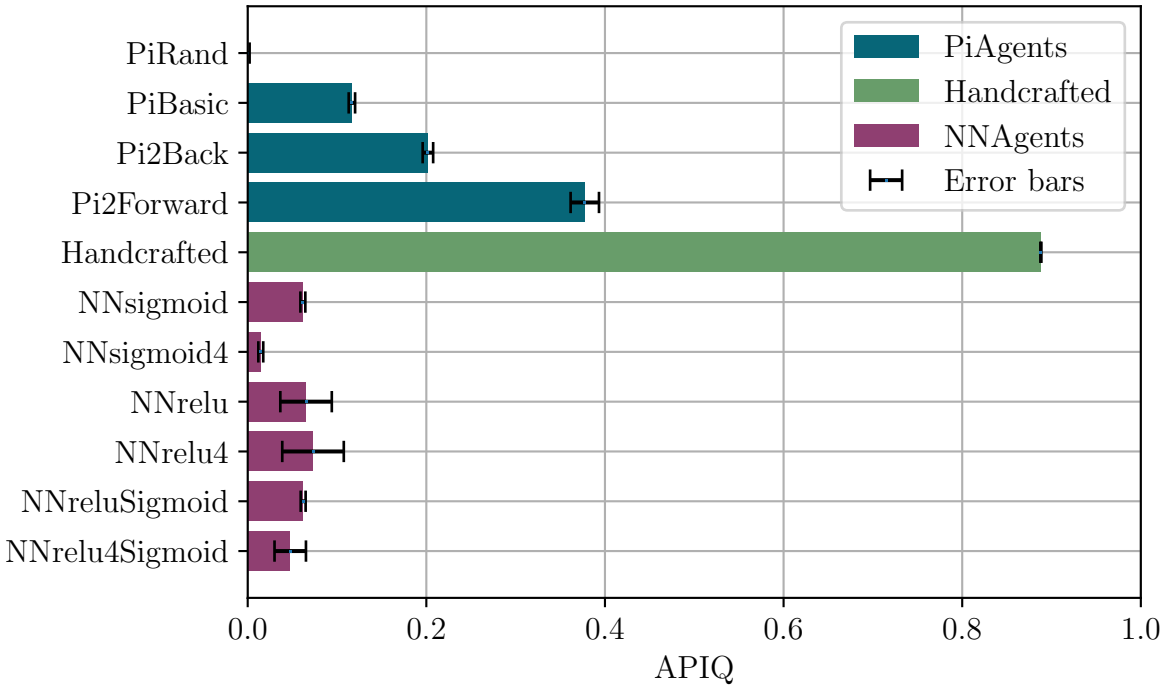

Figure 3: APIQ of all implemented agents

the rule, π^{basic} still performs better than π^{random} in most environments and therefore has higher APIQ than π^{random}.

Pi2Back (0.202 ± 0.006)**.** The agent which takes the last observation and action into account has similar scores to π^{basic} in most implemented environments. It does much better in Alternate which was designed to test for agents who take history into account and a little better in CombinationThree and LabyrinthLoop. Interestingly it does a little worse in AlternateRandomly, which probably is due to it managing a larger table and therefore being slower to react to the sudden changes when AlternateRandomly switches the action which returns reward. All things considered, π^{2back} performs better or equal to π^{basic} in all but one environment and as expected has higher APIQ.

Pi2Forward (0.377 ± 0.016)**.** $\pi^{2forward}$, the agent which takes the next 2 actions into account for its planning can exploit the regularities in Slide and SlideTwo and achieves positive reward for both. Other than that its trials have similar results to π^{2back}. Notable differences include it doing better in CombinationTwo, which requires the input of a 2 action combination for which $\pi^{2forward}$ is well suited and it doing slightly worse in DoubleCoinFlip where the random nature of the environment coupled with the larger table, that $\pi^{2forward}$ has to maintain, probably led to slower learning. Again as we expected, the theoretically superior agent $\pi^{2forward}$ has the highest APIQ of all implemented agents, which don't have previous knowledge about the environments.

NNsigmoid (0.062 ± 0.003)**.** NNsigmoid uses a simple neural network with no hidden layers and the sigmoid activation function for choosing its actions. It doesn't plan and doesn't take the history into account. Its APIQ is consequently very low but still higher than π^{rand}. As expected it has lower APIQ than π^{basic} since it doesn't keep statistics for every action-reward pair. In figure 5 we see that NNsigmoid does well in similar but fewer environments than π^{basic} and also makes the same mistake in Slide and SlideTwo.

NNsigmoid4 (0.015 ± 0.003)**.** NNsigmoid4 is similar to NNsigmoid, except that it uses a hidden layer of size 4. This seems to hinder it, rather than helping, since the agent achieves only about 1/4 the reward of NNSigmoid with similar variance. The received rewards for NNsigmoid4 look similar to NNsigmoid, but are always less in magnitude, even in the Slide environments.

NNrelu (0.065 ± 0.029)**.** NNrelu achieves higher mean reward than the sigmoid variants, but the variance is also about 10 times higher, which makes the relative error almost 50%. Although the overall APIQ is as expected on average lower than π^{basic} it achieves some reward in LabyrinthCoord where even the much more intelligent $\pi^{2forward}$ has none. LabyrinthCoord is an environment where the agent needs to get from a to b to receive reward and is given the coordinates as the observation. NNrelu also receives no negative reward Slide and much less in SlideTwo in contrast to the NNagents which use sigmoid. The high variance of the received reward for NNrelu also is very obvious in figure 5.

NNrelu4 (0.073 ± 0.034)**.** In spite of NNsigmoid4 having much lower APIQ than NNsigmoid, with the relu activation function having a hidden layer even slightly increases the APIQ to 0.073. However, it also increases the variance, causing the relative error to remain close to 50%. The breakdown by environment shows that NNrelu4 does better in some environments and worse in others but is overall very similar to NNrelu. It also suffers from the same high variance problem.

NNreluSigmoid (0.062 ± 0.003)**.** When we choose the popular combination of relu and sigmoid for our neural network, where sigmoid is only applied in the output layer, the APIQ and its variance are about the same as in NNsigmoid. When we take a closer look at the received rewards in figure 5, we see that the two agents' scores are almost identical.

NNrelu4Sigmoid (0.048 ± 0.018)**.** When the relu network with the sigmoid output layer gets an extra hidden layer of size 4, APIQ and variance are between NNSigmoid4 and ReluSigmoid4. Looking at the rewards, it is a mixture of NNsigmoid and NNrelu4. It is also the NNAgent with a hidden layer that does the worst in the Slide environments.

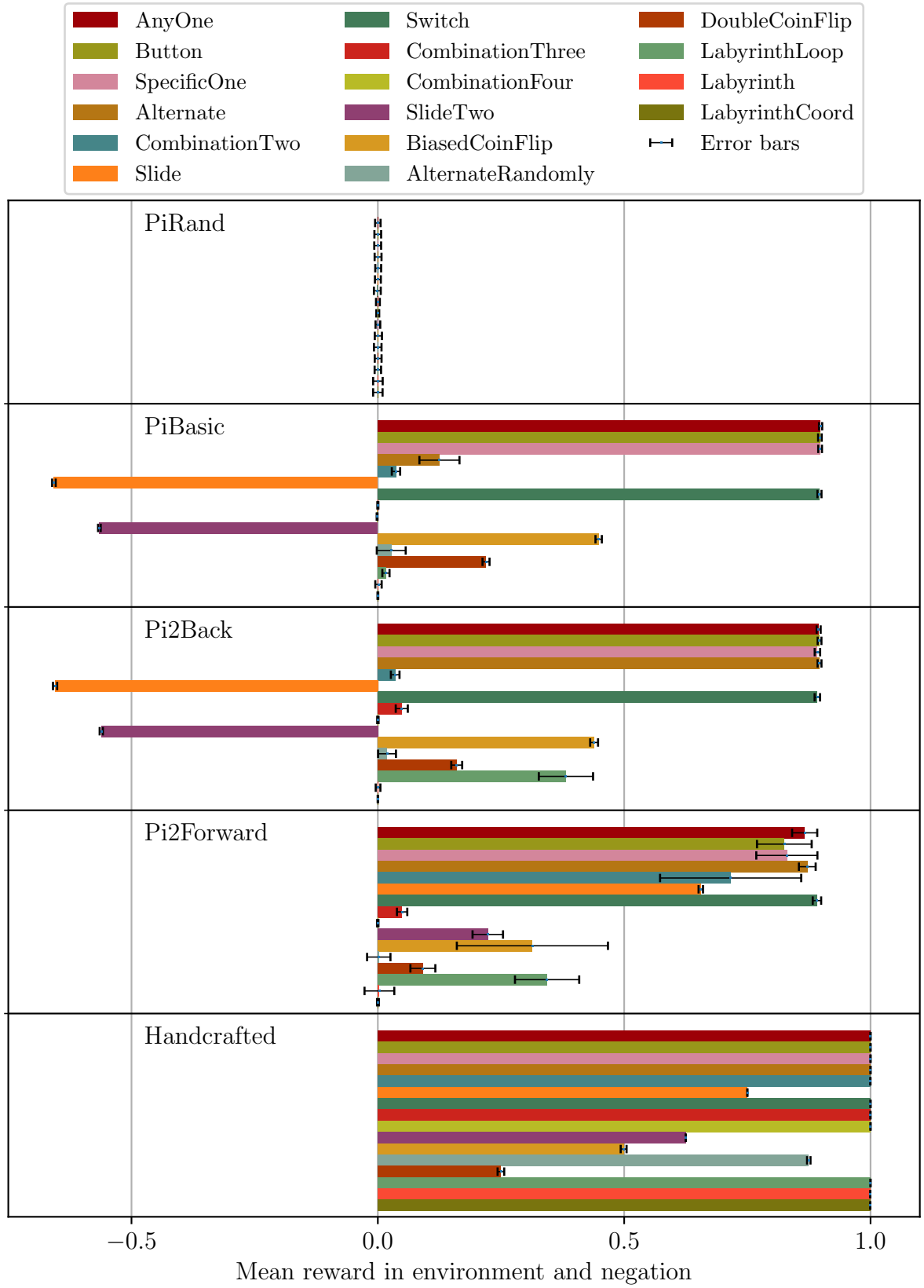

Figure 4: Results of the pi agents and the handcrafted agent being trialed in all environments and their negations.

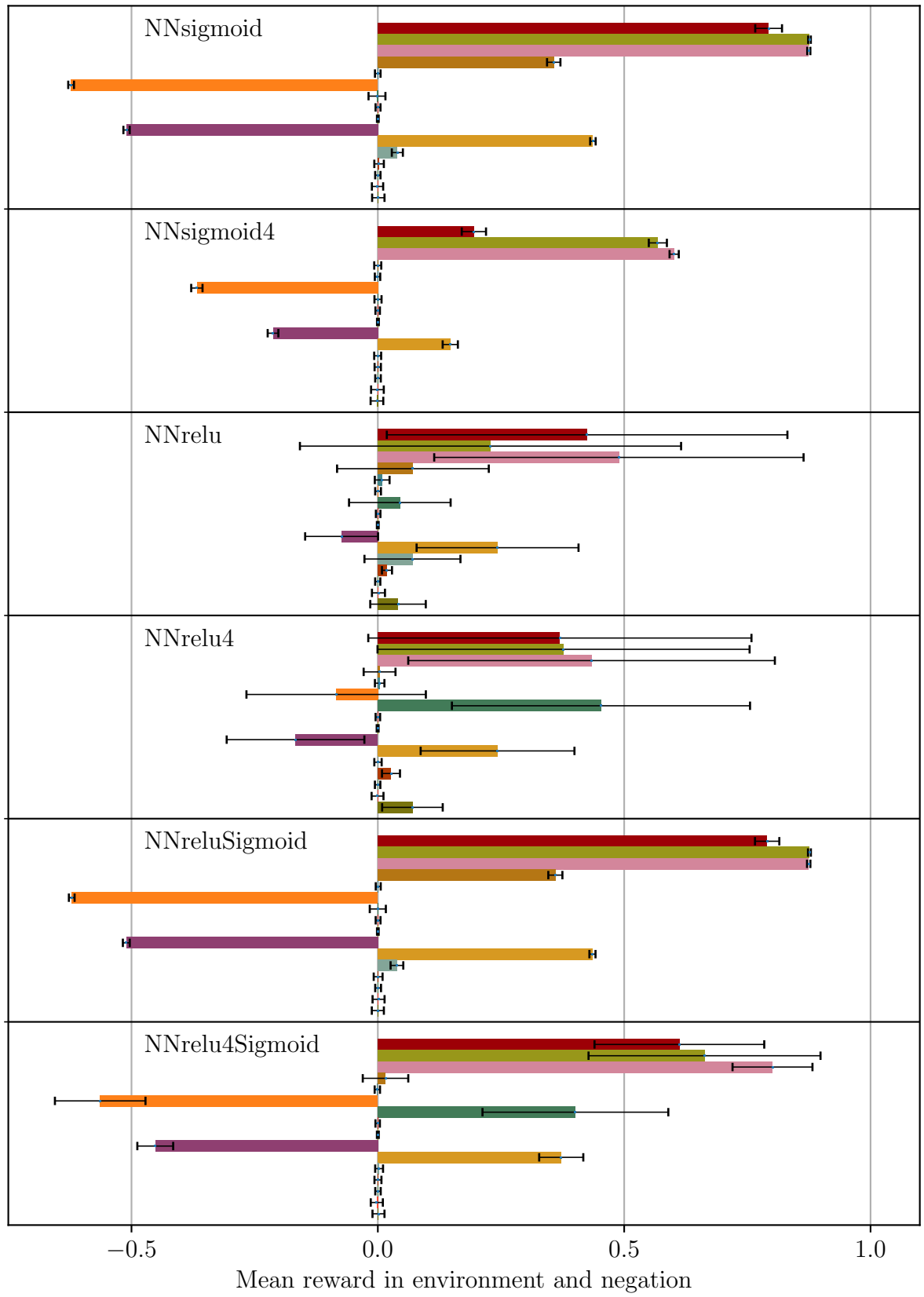

Figure 5: Results of the neural network based agents being trialed in all environments and their negations.

5 Discussion

5.1 Measure of Intelligence

The goal for this thesis is to create a measure of intelligence that approximates the UIM by Hutter and Legg [5]. The following is a listing of key points allowing us to argue that the goal has been achieved.

Ordering by Intelligence. The most important aspect of a measure of intelligence is that the intelligence it assigns to agents is valid. There should be no agent which is obviously inferior with a higher score than its superior agent. We have a clear ordering of intelligence in the pi agents discussed in the theory section 2 and the implementations of those agents behave as expected in the APIQ. Furthermore, the implemented simple neural network based agents behave as expected and achieve an APIQ between π^{rand} and π^{basic}. Given this limited data, we, therefore, conclude that APIQ orders agents sensibly, at least for obvious differences in their ability to achieve rewards in diverse environments. We expect the accuracy of the ordering to improve when more environments are added to the APIQ.

Complexity. Since Kolmogorov complexity is not computable, it is replaced by a count of the Bytecode instructions needed to implement an environment. There is no way to prove, that the environments have minimal complexity for non-trivial cases, but we assume that the difference between minimal and measured complexity is small for carefully implemented environments.

Occams razor. In the UIM, Occams razor is applied by weighting every environment with the *algorithmic prior distribution* $2^{-K(p)}$. If we just trialed one environment for every complexity, we wouldn't need the weighting, since every environment would be a sample of the $2^{K(p)}$ environments with the same complexity. Since we don't want to fulfill arbitrary complexity constrains on our implemented environments, we instead weight the environments inversely proportional by how many environments with similar complexity there are, dependent on the range of complexity covered and the number of implemented environments. This weighting will approach the algorithmic prior distribution when the number of implemented environments approaches infinity.

Generality. Although it has to make approximations, APIQ is still very general. An agent can be any entity able to work with bitstrings as inputs and outputs at sufficient speeds. Since an agent needs to calculate 32,000,000 actions to receive a score with the

current iteration of APIQ, humans can't be evaluated with it in its current form in a sensible timeframe.

5.2 Possible Improvements

Although the APIQ can be categorized as a measure of intelligence, there are still many obvious and less obvious ways to improve it. Itemizing them is also beneficial for recognizing its flaws.

Environments. Adding more environments for the agents to be trialed in is the easiest way to increase the accuracy and meaningfulness and decrease the variance of the measure. The complexity of added environments is preferred to be different from the existing ones. Here we run into two problems.

The first is simply computation speed. The more environments, the more trials have to be performed. It might then be beneficial to adjust the number of cycles in a trial or the number of trials to balance practicality and accuracy.

The second problem is inherent in our weighting method. If we add an environment with complexity much higher than the other environments, it will be weighted disproportionally. We, therefore, propose that an added environment cant have more than 120% of the complexity of the previous maximum complexity environment.

Complexity. The complexity is measured by counting the python bytecode instructions of the `calculate_percept` method that every environment has to implement. This limits the use of libraries since the functions are just called and their value returned, without their complexity being taken into account. For more complex environments that use libraries, a recursive method for counting bytecode instructions will be required.

Agents. Agents get trialed in environments, and need to have a fitting interface and speed, but aren't otherwise restricted. An interesting metric is the complexity of an agent. It allows us to order agents not only by intelligence but by the complexity it took to achieve this intelligence. This is much more difficult than with the environments since agents can be anything and one would either need some universal complexity or be content with only measuring the complexity of agents that are programs.

For agents like humans who can't process millions of interactions in a reasonable timeframe, we could drastically reduce the number of cycles for example to 20 and then sample from the pool of implemented environments for an evaluation. Having previous knowledge about the possible environments would give an unfair advantage, so either the environments

must be unknown or so numerous, that remembering them all is next to impossible.

Implementing more agents is another way of testing the APIQ more rigorously. It also is interesting to test the performance of different agents like in our neural network based agents, where using the relu activation function with a sigmoid output layer gives us the best of both worlds, namely better learning with lower variance.

6 Outlook

While the APIQ has no absolute claims on measuring universal intelligence, it sensibly orders simple agents. One important goal is for it to be used to evaluate learning algorithms, giving researchers valuable insight into their overall performance. The more accurate the evaluation, the more minute changes it can detect. It might even be possible to automate the evaluation and refinement of algorithms to a certain degree, giving algorithms a way to improve themselves.

More research and refinement is necessary for improvements and evaluation of the possible use cases of APIQ.

A A

A.1 environments

A.2 agents

References

[1] Adams, S., Arel, I., Bach, J., Coop, R., Furlan, R., Goertzel, B., Hall, J. S., Samsonovich, A., Scheutz, M., Schlesinger, M., Shapiro, S. C., and Sowa, J. Mapping the landscape of human-level artificial general intelligence. *AAAI Artificial Intelligence Magazine 33* (2012), 25–42.

[2] Hutter, M. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability.* Springer, Berlin, 2005.

[3] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[4] Legg, S., and Hutter, M. A collection of definitions of intelligence. In *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms* (Amsterdam, NL, 2007), B. Goertzel and P. Wang, Eds., vol. 157 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 17–24.

[5] Legg, S., and Hutter, M. Universal intelligence: A definition of machine intelligence. *Minds & Machines 17*, 4 (2007), 391–444.

[6] Legg, S., and Veness, J. An approximation of the universal intelligence measure. *CoRR* (2011).

[7] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature 529* (2016), 484–503.

[8] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR abs/1712.01815* (2017).

[9] Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T. P., Calderone, K., Keet, P., Brunasso, A.,

Lawrence, D., Ekermo, A., Repp, J., and Tsing, R. Starcraft II: A new challenge for reinforcement learning. *CoRR abs/1708.04782* (2017).

Assertion

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, July 9, 2020 Christian Hauser