Extending Environments To Measure Self-Reflection In Reinforcement Learning

Samuel A. Alexander*
Michael Castaneda†
Kevin Compher‡
Oscar Martinez§

October 13, 2021

Abstract

We consider an extended notion of reinforcement learning in which the environment can simulate the agent and base its outputs on the agent's hypothetical behavior. Since good performance usually requires paying attention to whatever things the environment's outputs are based on, we argue that for an agent to achieve on-average good performance across many such extended environments, it is necessary for the agent to self-reflect. Thus, an agent's self-reflection ability can be numerically estimated by running the agent through a battery of extended environments. We are simultaneously releasing an open-source library of extended environments to serve as proof-of-concept of this technique. As the library is first-of-kind, we have avoided the difficult problem of optimizing it. Instead we have chosen environments with interesting properties. Some seem paradoxical, some lead to interesting thought experiments, some are even suggestive of how self-reflection might have evolved in nature. We give examples and introduce a simple transformation which experimentally seems to increase self-reflection.

1 Introduction

An obstacle course might react to what you do: for example, if you step on a certain button, then spikes might appear. If you spend enough time in such an obstacle course, you should eventually figure out such patterns. But imagine an "oracular" obstacle course which reacts to what you would hypothetically do in counterfactual scenarios: for example, there is no button, but spikes appear if you

*The U.S. Securities and Exchange Commission, samuelallenalexander@gmail.com
†KX
‡The U.S. Securities and Exchange Commission
§The U.S. Securities and Exchange Commission

would hypothetically step on the button if there was one. Without self-reflecting about what you would hypothetically do in counterfactual scenarios, it would be difficult to figure out such patterns. This suggests that in order to perform well (on average) across many such obstacle courses, some sort of self-reflection is necessary.

This is a paper about empirically estimating the degree to which a Reinforcement Learning (RL) agent is self-reflective. By a self-reflective agent, we mean an agent which acts not just based on environmental rewards and observations, but also based on considerations of its own hypothetical behavior. We propose that an RL agent's degree of self-reflection can be estimated by running the agent through a battery of environments which we call *extended environments*, environments which react not only to what the agent does but to what the agent would hypothetically do. For good performance averaged over many such environments, an agent would need to self-reflect about itself, because otherwise, environment responses which depend on the agent's own hypothetical actions would often seem random and unpredictable. The extended environments which we consider are a departure from standard RL environments, but this does not interfere with their usage for judging standard RL agents: one can run a standard agent in an extended environment in spite of the latter's non-standardness.

To understand why extended environments (where the environment reacts to what the agent would hypothetically do) incentivize self-reflection, consider a game involving a box. The contents of the box change from playthrough to playthrough, and the game's mechanics depend upon those contents. The player may optionally choose to look inside the box, at no cost: the game does not change its behavior based on whether the player looks inside the box. Clearly, the player has an incentive to look inside the box. The extended environments we consider are similar to this example. Instead of a box's contents, the game's mechanics depend upon the player. Rather than looking into a box, the environment "looks into" the player (by simulating a copy of the player) and adjusts its mechanics accordingly. Just like the player in the above game is incentivized to look in the box, an agent designed to perform in extended environments is incentivized to examine itself, that is, to self-reflect.

One might try to imitate an extended environment with a non-extended environment by backtracking—rewinding the environment itself to a prior state after seeing how the agent performs along one path, and then sending the agent along a second path. But the agent itself would retain memory of the first path, and the agent's decisions along the second path might be altered by said memories. Thus the result would not be the same as immediately sending the agent along the second path while secretly simulating the agent to determine what it would do if sent along the first path.

Alongside this paper, we are publishing an MIT-licensed open-source library [2] of extended environments to "ease adoption by other machine-learning researchers" [31]. We are inspired by similar (but non-extended) libraries and other benchmark collections [4] [5] [6] [8] [9] [12] [24] [33]. Our library is intended show that it is possible to numerically estimate the self-reflectiveness of RL agents. Aside from measuring self-reflectiveness of individual agents, such a benchmark can also be used to experimentally test agent transformations intended to make agents more self-reflective (we introduce one such transformation in Section 5).

Our benchmark is based on Legg and Hutter's theory of universal intelligence measurement [18]. Legg and Hutter argue that to perfectly measure RL agent performance, one should aggregate the agent's performance across the whole space of all sufficiently well-behaved environments, weighted using an appropriate distribution. Rather than a uniform distribution (susceptible to no-free-lunch theorems), Legg and Hutter suggest assigning more weight to simpler environments and less weight to more complex environments. Thus, a high-order approximation of Legg and Hutter's idealized measure would need to involve representative environments of several complexities, weighted accordingly. But measuring these complexities is hard and subjective [20], and we make no attempt to do so. Our library is intended as a rough first-order approximation, using only $n = 25$ simple extended environments, each with weight $1/n$ (all other environments are considered to have weight $0 \neq 1/n$, so the distribution is non-uniform and no-free-lunch does not apply). In choosing those n simple extended environments, we have sought environments interesting in their own right[1], exhibiting paradoxes, interesting thought experiments, counter-intuitive winning strategies, or even shedding light on how self-reflection might have evolved in nature. We will discuss examples in Section 3.

2 Preliminaries

A formal, theoretical treatment of RL is necessary to make the mathematics behind extended environments clear. This formal version of RL differs significantly from how RL is implemented in practice. In our library [2] we implement a more realistic and practical formalization (including non-deterministic agents and environments). We discuss the more practical formalization in Section 4.

Our formal treatment of RL is based on Section 4.1.3 of [17], except that we assume the agent receives an initial percept before taking its initial action (whereas in [17], the agent acts first). We will write $x_1y_1\ldots x_ny_n$ for the length-$2n$ sequence $\langle x_1, y_1, \ldots, x_n, y_n\rangle$ and $x_1y_1\ldots x_n$ for the length-$(2n-1)$ sequence $\langle x_1, y_1, \ldots, x_n\rangle$. In particular when $n = 1$, we will write $x_1y_1\ldots x_n$ for $\langle x_1\rangle$, even if y_1 is not defined. We assume fixed finite sets of actions and observations. By a *percept* we mean a pair $x = (r, o)$ where o is an observation and $r \in \mathbb{Q}$ is a reward.

Definition 1. *(RL agents and environments)*

1. *A (non-extended)* environment *is a function μ which outputs an initial percept $\mu(\langle\rangle) = x_1$ when given the empty sequence $\langle\rangle$ as input and which, when given a sequence $x_1y_1\ldots x_ny_n$ as input (where each x_i is a percept and each y_i is an action), outputs a percept $\mu(x_1y_1\ldots x_ny_n) = x_{n+1}$.*

[1] We would compare our library with benchmarks like the ubiquitous Atari benchmark [4]. Arguably that benchmark uses the same strategy, choosing its n environments not based on theoretical interest but on which video-games were developed and marketed for the Atari 2600.

2. *An* agent *is a function* π *which, given a sequence* $x_1y_1 \dots x_n$ *as input (each* x_i *a percept, each* y_i *an action), outputs an action* $\pi(x_1y_1 \dots x_n) = y_n$*.*

3. *If* π *is an agent and* μ *is an environment, the* result of π interacting with μ *is the infinite sequence* $x_1y_1x_2y_2 \dots$ *defined by:*

$$
\begin{array}{ll}
x_1 = \mu(\langle\rangle) & y_1 = \pi(\langle x_1\rangle) \\
x_2 = \mu(\langle x_1, y_1\rangle) & y_2 = \pi(\langle x_1, y_1, x_2\rangle) \\
x_3 = \mu(\langle x_1, y_1, x_2, y_2\rangle) & y_3 = \pi(\langle x_1, y_1, x_2, y_2, x_3\rangle) \\
\quad \dots & \quad \dots \\
x_n = \mu(x_1y_1 \dots x_{n-1}y_{n-1}) & y_n = \pi(x_1y_1 \dots x_n) \\
\quad \dots & \quad \dots
\end{array}
$$

In the following definition, we extend environments by allowing their outputs to depend not only on $x_1y_1 \dots x_ny_n$ but also on an agent π. Intuitively, extended environments are environments with the ability to simulate the agent[2].

Definition 2. *(Extended environments)*

1. *An* extended environment *is a function* μ *which outputs initial percept* $\mu(\pi, \langle\rangle) = x_1$ *in response to input* $(\pi, \langle\rangle)$ *where* π *is an agent; and which, when given input* $(\pi, x_1y_1 \dots x_ny_n)$ *(where* π *is an agent, each* x_i *is a percept and each* y_i *is an action), outputs a percept* $\mu(\pi, x_1y_1 \dots x_ny_n) = x_{n+1}$*.*

2. *If* π *is an agent and* μ *is an extended environment, the* result of π interacting with μ *is the infinite sequence* $x_1y_1x_2y_2 \dots$ *defined by:*

$$
\begin{array}{ll}
x_1 = \mu(\pi, \langle\rangle) & y_1 = \pi(\langle x_1\rangle) \\
x_2 = \mu(\pi, \langle x_1, y_1\rangle) & y_2 = \pi(\langle x_1, y_1, x_2\rangle) \\
x_3 = \mu(\pi, \langle x_1, y_1, x_2, y_2\rangle) & y_3 = \pi(\langle x_1, y_1, x_2, y_2, x_3\rangle) \\
\quad \dots & \quad \dots \\
x_n = \mu(\pi, x_1y_1 \dots x_{n-1}y_{n-1}) & y_n = \pi(x_1y_1 \dots x_n) \\
\quad \dots & \quad \dots
\end{array}
$$

The reader might notice that it is superfluous for μ to depend both on π and $x_1y_1 \dots x_ny_n$ since, given just π and n, one can reconstruct $x_1y_1 \dots x_ny_n$. We intentionally choose the superfluous definition because it better captures our intuition (and makes clear the similarity to Definition 1). For the sake of simpler mathematics, we have not included non-determinism in our formal definition, but in practice, agents and environments are often non-deterministic, so that π and n do not determine $x_1y_1 \dots x_ny_n$ (our practical treatment in [2], discussed in Section 4, does allow non-determinism).

[2]This can be considered a dual version of AIs which simulate their environment, as in Monte Carlo Tree Search [7].

The fact that classical agents can interact with extended environments (Definition 2 part 2) implies that various universal RL intelligence measures [1] [11] [13] [10] [18] [19], which measure performance in (non-extended) environments, easily generalize to measure self-reflective intelligence (performance in extended environments)[3]. For example, Legg and Hutter's [18] universal intelligence measure $\Upsilon(\pi)$ is defined to be agent π's average reward-per-environment, aggregated over all (non-extended) environments with suitably bounded rewards, each environment being weighted using the algorithmic prior distribution [21]. Simply by including suitably reward-bounded extended environments, we immediately obtain a variation $\Upsilon'(\pi)$ which measures the performance of π across extended environments—apparently a version of Legg and Hutter's universal intelligence which measures both intelligence and self-reflection ability.

3 Some interesting extended environments

In this section, we exhibit some interesting examples of extended environments.

3.1 A quasi-paradoxical extended environment

Example 3. *(Rewarding the Agent for Ignoring Rewards) For every percept $x = (r, o)$, let $x' = (0, o)$ be the result of zeroing the reward component of x. Fix some observation O. Define an extended environment μ as follows, where $x_1y_1 \ldots x_ny_n$ ranges over percept-action sequences ending in an action:*

$$\mu(\pi, \langle\rangle) = (0, O),$$
$$\mu(\pi, x_1y_1 \ldots x_ny_n) = \begin{cases} (1, O) & \text{if } y_n = \pi(x'_1y_1 \ldots x'_n), \\ (-1, O) & \text{otherwise.} \end{cases}$$

In Example 3, every time the agent takes an action y_n, μ simulates the agent in order to determine: would the agent have taken the same action if the history so far were identical except for all rewards being 0? If so, then μ gives the agent $+1$ reward, otherwise, μ gives the agent -1 reward. Thus, the agent is rewarded for ignoring rewards. Example 3 seems paradoxical: suppose an agent guesses the pattern and begins deliberately ignoring rewards, so long as the rewards it receives for doing so remain consistent with that guess. In that case, does the agent ignore rewards, or not? The paradox can be summarized: "I ignore rewards because I'm rewarded to do so."

Example 3 is implemented in [2] as IgnoreRewards.py. A key strength of the formalism in Definition 2 is that by explicitly defining an extended environment, as in Example 3, we avoid ambiguity inherent in everyday language. If one merely said informally, "reward the agent for ignoring rewards", that could be interpreted in various different ways. To show this, we implement two other interpretations

[3] To quote Legg and Veness: "Having a suite of such tests, with each emphasizing different, measurable aspects of intelligence, would clearly help the community build more powerful and robust general agents" [19].

as IgnoreRewards2.py (which interprets the English phrase by reading "reward" as "positive reward", i.e., as opposed to "punishment") and IgnoreRewards3.py (which interprets the phrase as "reward agent if agent's nth action equals the nth action the agent would take in the all-reward-0 environment").

3.2 A counterintuitive winning strategy

Example 4. *(Tempting Button) Fix an observation B (thought of as "there is a button") and an action A ("push the button"), and assume there is at least one other observation and one other action. For each percept-action sequence $h = x_1y_1 \ldots x_n$, if the observation in x_n is not B, then let h' be the sequence equal to h except that the observation in x_n is replaced by B. Let $o_0, o_1, o_2, \ldots$ be observations generated pseudo-randomly such that for each i, $o_i = B$ with 25% probability and $o_i \neq B$ with 75% probability. Let $\mu(\pi, \langle\rangle) = (0, o_0)$, and for each percept-action sequence $h = x_1y_1 \ldots x_n$ and action y_n, define $\mu(\pi, h \frown y_n)$ as follows (where O is the observation in x_n):*

$$\mu(\pi, h \frown y_n) = \begin{cases} (1, o_n) & \text{if } O = B \text{ and } y_n = A; \\ (-1, o_n) & \text{if } O = B \text{ and } y_n \neq A; \\ (-1, o_n) & \text{if } O \neq B \text{ and } \pi(h') = A; \\ (1, o_n) & \text{if } O \neq B \text{ and } \pi(h') \neq A. \end{cases}$$

In Example 4, if we think of the agent wandering from room to room:

- Each room either has a button (with 25% probability) or does not have a button (75% probability).
- In a room with a button, the agent gets +1 reward for pushing the button, −1 reward for not pushing it.
- In a room with no button, it does not matter what the agent does. The agent is rewarded or punished based on what the agent *would* do if there *was* a button. If the agent *would* push the button (if there was one), then the agent gets reward −1. Otherwise, the agent gets reward +1.

Thus, whenever the agent sees a button, the agent can push the button for a free reward with no consequences presently nor in the future; nevertheless, it is in the agent's best interest to commit itself to never push the button! Pushing every button yields an average reward of $1 \cdot (.25) - 1 \cdot (.75) = -.5$ per turn, whereas a policy of never pushing the button yields an average reward of $-1 \cdot (.25) + 1 \cdot (.75) = +.5$ per turn.

In Example 4, when the environment simulates the agent in order to determine whether the agent would press the button if there was a button, the true agent is not altered by the simulation. If the agent's actions are based on a Q table or a neural net, the simulation will include a simulation of that Q table or neural net, and that simulated Q table or neural net might be updated, but the true agent's Q table or neural net is not directly updated by the simulation. Thus, unless the

Table 1: Performance in Example 4 (100k steps)

Agent	Avg Reward-per-turn ± StdErr (test repeated with 5 RNG seeds)
Q	-0.44858 ± 0.00044
DQN	-0.46687 ± 0.00137
A2C	-0.49820 ± 0.00045
PPO	-0.24217 ± 0.00793

agent itself introspects about its own hypothetical behavior ("What would I do if there was a button here?"), it seems the agent would have no way of realizing that the rewards in buttonless rooms depend on said behavior. Indeed, in Table 1 we see that industry-standard agents perform poorly in Example 4 (these numbers are extracted from result_table.csv in [2], which contains performance details for these and other agents on all the environments in our benchmark; see Sections 4 and 6 for more implementation details).

Example 4 is implemented in our open-source library as TemptingButton.py.

3.3 An interesting thought experiment

Example 5. *(Reverse history) Fix some observation O. Define an extended environment μ as follows:*

$$\mu(\pi, \langle\rangle) = (0, O),$$

$$\mu(\pi, x_1y_1\ldots x_ny_n) = \begin{cases} (1, O) & \text{if } y_n = \pi(x_ny_{n-1}x_{n-1}y_{n-2}\ldots y_1x_1), \\ (-1, O) & \text{otherwise.} \end{cases}$$

In Example 5, whenever the agent takes an action y_n, μ simulates the agent in order to determine: would the agent have taken that same action if everything earlier had happened in reverse? If so, reward the agent, otherwise, punish the agent. Thus, the agent is rewarded for acting the same way that it would act if time were reversed.

It is interesting to think about what it would be like to interact with the environment in Example 5. It would be like wearing a shock collar which shocks one whenever one acts differently than one would act if time were moving backwards. To approximate the experiment, a test subject, commanded to speak backwards, might be constantly rewarded or punished for obeying or disobeying. This might teach the test subject to imitate backward speech, but then the test subject would still act as if time were moving forward, only they would do so while performing backward-speech (they would hear their own speech backwards). But if the experimenter could perfectly simulate the test subject in order to determine what the test subject would do if time really was moving backwards, what would happen? Could test subjects learn to behave as if time was reversed[4]? For example, by forward-speaking a monologue of a person suffering a backwards-speech hallucination? ("Hello. Hey, why does my voice sound so weird?!") Another possibility is that humans might simply not be capable of performing well in the environment. Our self-reflectiveness measure is not intended to be limited to human self-reflection levels.

We implement Example 5 as ReverseHistory.py in [2].

3.4 Additional examples in brief

Here are a few additional extended environment examples, without full details. We indicate in parentheses where these environments are implemented in [2].

- (SelfRecognition.py) Environments which reward the agent for recognizing actions it itself would take. We implement an environment where the agent sees observations which encode True-False statements such as "If this observation were 0, you would take action 1," and so on, and the agent is rewarded for deciding whether those statements are true or false.
- (IncentivizeLearningRate.py) Environments which reward the agent for behaving as if the agent were configured with a particular learning rate (suggesting that extended environments can incentivize agents to learn about their own internal mechanisms, as in [29]). Note that this requires a more practical RL formalization than that of Definition 2; we discuss the more practical formalization in Section 4.
- (AdversarialSequencePredictor.py) Environments which pit the agent against a competitor in an adversarial sequence prediction game [15]. This is done by outsourcing the competitor's behavior to the agent's own action-function (a technique we will explore in more detail in Example 8), thus avoiding the need to hard-code a competitor into the environment.
- (RuntimeInspector.py) Environments which reward the agent for responding quickly. In our implementation, we use Python's debugger (pdb) to count the number of steps the agent takes to act.
- (DeterminismInspector.py) Environments which reward the agent for being deterministic. In our implementation, the environment calls the agent's action-function redundantly to check whether it returns the same action.

4 Extended Environments in Practice

Definitions 1 and 2 are computationally impractical if agents are to run on environments for many steps. Even just maintaining long enough percept-action sequences can be prohibitive, not to mention computing actions and percepts from

[4]The difference between behaving as if the incentivized experience were its experience and actually experiencing that as its real experience brings to mind the objective misalignment problem presented in [16].

those sequences. In this section, we will discuss a more practical formalization of extended environments. Our reasons for doing this are fourfold:

1. The more practical implementation allows our library to run quickly enough that it becomes feasible to run industry-standard agents against it for many steps. This is important because most industry-standard agents require many steps to learn the environments they are placed in.

2. Certain environments have much clearer formulations using the practical framework. They could still be formulated using the theoretical framework, but doing so would involve lots of tedious mathematical notation.

3. We find it interesting in its own right how certain environments can be implemented in a practical way whereas others apparently cannot.

4. Non-determinism is effortless and natural in the practical implementation.

In actual practice, RL works more like the following.

- An environment responds to individual actions rather than to histories, but while doing so, it may update aggregate historical information in its internal memory (e.g., in the fields of a Python class instance).

- An agent has two fundamental operations. First, it chooses actions in response to individual observations (not histories). These actions may also depend on the contents of the agent's internal memory[5] (e.g., the fields of a Python class instance, which may include Q tables, neural network weights, and so on). The agent does not update its internal memory during the process of acting. The agent only updates its internal memory during the second fundamental operation, *training*[6]. The agent trains in response to a *transition*, consisting of a previous observation, an action, a reward, and a next observation (the intuition being that the agent took the action in response to the previous observation, received the reward for doing so, and observed the next observation afterward). During training, the agent may update its internal memory (for example, updating neural network weights, adding the transition to a limited set of experiences for experience replay [23], and so on).

Because of how practical RL agents' actions depend on internal memory, some care is needed in order to make extended environments practical. In order for an environment to ask, "What would the agent do in response to such-and-such hypothetical history?", the environment would have to *train* the agent with

[5]Usually, a practical agent's total memory usage will be bounded, if not outright constant; in fact, even the runtime of the agent's action-function is often bounded or constant. A practical agent attempts to "behave as well as possible given its computational resources" [28].

[6]Indeed, it is common to pre-train a general-purpose agent on a particular environment—"a single function in isolation" [32]—and publish the resulting action-function as a special-purpose solution exclusively for that one environment, with the no-longer-needed training operation stripped away.

such-and-such hypothetical history (being unable to simply feed the hypothetical history to the agent's action-function as in Examples 3–5). But training the actual agent would alter the agent's internal memory, an unintended side effect which would compromise the whole idea of extended environments.

Thus, to practically realize extended environments in [2], rather than passing the environment an agent, we pass the environment an agent-class (or agent-factory) which may be used to create untrained copies of the agent, called *instances* of the agent-class. This actually turns out to be quite convenient because many industry-standard RL agents really do come from such factories (e.g., Python classes which one instantiates to get an actual agent). The environment can use the agent-class to create as many copies of the agent as needed, storing them in its internal memory. Since agents' action-functions are functions of single observations, not entire histories, environments such as Example 3 must instead *train* simulated agents with said histories. Of course, to train a fresh new simulated agent with an entire history at every step would be impractical. This can often be avoided by maintaining a simulated agent and training it gradually: there is no need to repeat trainings performed earlier, if the trained simulation is still available. To illustrate this, here is a practical version of Example 3.

Example 6. *The following Python code is a practical version of Example 3.*

```
class IgnoreRewards:
  def __init__(self, A):
    # Calling A() creates agent-copies. On
    # initiation, this environment stores
    # one such copy in its internal memory.
    self.sim = A()
  def start(self):
    return 0  # Initial observation 0
  def step(self, action):
    # At each step, use the stored copy
    # (self.sim) to determine how the true
    # agent would behave if all history so
    # far were the same except all rewards
    # were 0. Assumes self.sim has been
    # trained the same as the true agent,
    # except with all rewards 0.
    hypothetical_act = self.sim.act(obs=0)
    reward = 1 if action==hypothetical_act\
        else -1
    # To maintain above assumption, train
    # self.sim as if current reward were 0.
    # True agent will automatically train
    # the same way with the true reward.
    self.sim.train(o_prev=0, a=action,
        r=0, o_next=0)
    return (reward, 0)  # Observation=0
```

Unfortunately, not all extended environments (as in Definition 2) can be realized in a practical way. But surprisingly many of them can be. We originally wrote our library [2] to directly implement a variation of Definition 2, but the library was slow. To make environments practical, we had to convert them to the more practical framework, replacing expensive computations $\pi(x_1y_1\ldots x_n)$ with gradual, reusable training, as in Example 6. We feared that for many environments, this would not be possible. But it turned out to be possible for 23 out of 25 environments. The two exceptions were the Reverse History environment (Example 5) and an environment based on *Déjà Vu*.

The reason Example 5 is inherently impractical is because there is no way for the environment to re-use its previous work to speed up its next percept calculation. Even if the environment retained a simulated agent trained on the previous reverse history $h_0 = x_{n-1}y_{n-2}\ldots y_1x_1$, in order to compute the next percept, the environment would need to *insert* a new percept-action pair x_ny_{n-1} at the *beginning* of h_0 to get the new reverse history $h = x_ny_{n-1}\ldots y_1x_1$. There is no guarantee that the agent's actions are independent of the order in which it is trained, so a fresh new agent simulation would need to be created and trained on all of h from scratch. Similar remarks go for the *Déjà Vu* environment, which involves inserting new percept-action pairs in the *middle* of the previous training sequence. Our library retains ReverseHistory.py and DejaVu.py, but marked as "slow" (to exclude them from the benchmark); we replaced them with other environments to keep the benchmark size equal to 25.

4.1 Newcomb's Paradox

We find it insightful to use the framework we have developed to formalize a variation of Newcomb's paradox [25]. This is a well-known paradox in which a player is confronted with two boxes, a transparent box which visibly contains \$1000 and an opaque box which may contain a million dollars (\$1M) or nothing. The player must choose whether to take both boxes, or to only take the opaque box. The player knows that the opaque box's contents have been determined by a reliable predictor, who predicted what the player would do: if the predictor predicted the player will take both boxes, then the predictor put nothing in the opaque box; but if the predictor predicted the player will take just the opaque box, then the predictor put \$1M in the opaque box. In describing the paradox, Nozick reported that "To almost everyone, it is perfectly clear and obvious what should be done. The difficulty is that these people seem to divide almost evenly on the problem, with large numbers thinking that the opposing half is just being silly" [25].

The paradox does not exactly fit within reinforcement learning because it involves the player having *knowledge*, whereas an RL agent merely acts and trains and does not officially *know* anything (it can act as if it knows things, but it seems to us that the whole point of Newcomb's paradox is that no-one agrees what exactly it means to act as if knowing what the player in Newcomb's paradox knows). Thus, for RL, it is more appropriate to imagine the player playing the game over and over again many times, and instead of assuming that the player *knows* anything, we consider that the player could potentially infer patterns in the game.

Example 7. *(Newcomb's Paradox) The following Python code formalizes an RL variation of Newcomb's paradox.*

```
TAKE_ONE_BOX, TAKE_BOTH_BOXES = 0, 1
OBS = 0  # Opaque box + transparent box w/$1000

class NewcombEnvironment:
  def __init__(self, A):
    self.sim = A()  # Create copy of agent

  def start(self):
    predicted_action = self.sim.act(OBS)
    if predicted_action == TAKE_BOTH_BOXES:
        self.opaque_contents = 0
    else:
        self.opaque_contents = 1000000
    return OBS  # Show player two boxes

  def step(self, action):
    if action == TAKE_BOTH_BOXES:
      reward = 1000 + self.opaque_contents
    else:
      reward = self.opaque_contents

    self.sim.train(o_prev=OBS,
      a=action, r=reward, o_next=OBS)
    self.start()  # Repopulate opaque box

    return (reward, OBS)
```

In Example 7, although the environment keeps track of the contents of the opaque box, those are not shown to the agent—the box is opaque. Thus, the agent always sees the same observation. It is an example of an RL environment in which there is state that is not visible to the agent. One of the key strengths of our framework is that by formalizing environments concretely as in Example 7, ambiguities in natural language are annihilated. Thus, it is unambiguous that if A_1 is an agent-class whose instances always take the opaque box only, then A_1's instances will get 1000000 reward per turn in Example 7. If A_2 is an agent-class whose instances always take both boxes, then A_2's instances will get 1000 reward per turn. Of course, we do not claim that this solves the original Newcomb's paradox, since the original paradox involves *knowledge*, which does not fit in the RL framework. In fact, interestingly, when we run standard RL agents on Example 7, the subtle nuances of Newcomb's Paradox are lost on them: despite having no knowledge of how the environment works, they quickly learn that it's better to take only the opaque box.

4.2 An extended environment of biological interest

> "It is only when people are embedded in a complex competitive social environment that the goal of interacting with others requires them to anthropomorphise their own actions. This recursive modelling

gives rise to an understanding of selfhood, an appreciation of the first-person experiential self."—Maguire et al [22]

Extended environments offer an elegant solution to a difficult problem that arises in measuring RL intelligence. The problem, described in [14], is that "the probability of other agents appearing on the scene or having some social interaction is almost 0": building a sophisticated competitor (or collaborator) RL agent into a standard RL environment would increase the complexity of that environment (because the environment would include said competitor agent's source-code). Thus, intelligence measures based on giving more weight to simpler environments would give little weight to environments with sophisticated built-in RL agents. Such measures might therefore fail to capture social aspects of intelligence. The following example shows that a relatively simple extended environment can include other agents of arbitrary complexity, by outsourcing those other agents' actions to the agent whose intelligence is being measured.

Example 8. *(Crying Baby) The following code defines an environment which consists of a baby, and the agent must decide when to feed the baby. The agent is rewarded when the baby laughs, punished when the baby cries. The baby's behavior (whether to laugh or cry) is computed by simulating the agent to determine what the agent would do if the agent were in the baby's position.*

```
FEED, DONTFEED = LAUGH, CRY = 0, 1
class CryingBaby:
  def __init__(self, A):
    # Obtain baby by simulating agent
    self.baby = A()
    self.full = 5  # Nutrition
    self.prev_baby_obs = FEED
  def start(self):
    return LAUGH  # First obs is laughter
  def step(self, action):
    if action == FEED:
      self.full = min(self.full+1, 9)
    else:
      self.full = max(self.full-1, 0)
    baby_act = self.baby.act(obs=action)
    parent_reward = 1 if (baby_act==LAUGH)\
        else -1
    baby_reward = 1 if (2 < self.full < 8)\
        else -1
    parent_obs, baby_obs = baby_act, action
    self.baby.train(
      o_prev=self.prev_baby_obs,
      a=baby_act, r=baby_reward,
      o_next=baby_obs)
    self.prev_baby_obs = baby_obs
    return (parent_reward, parent_obs)
```

Obviously, Example 8 is a gross over-simplification of parent-child dynamics, but it illustrates how multi-agent non-extended environments can be transformed into single-agent extended environments, by letting the same agent play all the different roles. One might be tempted to visualize a bunch of identical twins interacting, but that would be misleading. To quote Silver et al: "The agent consists solely of the decision-making entity; anything outside of that entity (including its body, if it has one) is considered part of the environment" [30]. Thus in Example 8, when an A-instance plays the parent's role, its actions control the parent's body. When another A-instance plays the baby's role, its actions control the baby's body. The environment could even be modified to include decades worth of turns before the baby arrives, to capture the realistic fact that a parent has more life experience than a baby (and acts accordingly), even while still using the same agent-class for both roles.

With the above in mind, extended environments might shed light on how living organisms evolve self-reflection. Assume that descendants' policy source-codes are approximately equal to their recent ancestors' policy source-codes. Then whenever an organism interacts with similar organisms, it interacts with an environment whose reactions depend (via those other organisms' actions) approximately on that organism's own source-code. The closer the organism is related to the other organisms with which it interacts, the better the approximation. A human interacting with another human might achieve better results by self-reflectively considering, "What would I do in this other person's position?"

The idea behind Example 8 would easily generalize to allow relatively simple single-agent extended environments equivalent to multi-agent environments with many, possibly highly complicated, interacting agents (even a whole society of interacting agents, who "define their own rewards and punishments because they themselves assist in the rewards and punishments" [26]).

We implement Example 8 as CryingBaby.py in [2].

4.3 Determinacy and Semi-Determinacy

For the sake of simpler mathematics, we have been formally working in a deterministic version of RL. This simplifies the mathematics because mathematical functions are deterministic by definition. Functions in the computer-programming sense have no determinism requirement: their output may depend on random number generators (RNGs), time-of-day, global variables, system calls, 3rd-party libraries, or even on responses from a remote server or from a human being. Our proposed method of measuring agent self-reflection could still be applied even to non-deterministic agents: run the agent on a battery of extended environments and estimate its self-reflection by averaging the resulting rewards. Depending on the nature of the non-determinism, this measurement might not make much sense. For example, if an agent operates by reading and writing files on disk, then a simulation of that agent might influence the true agent (by altering said files), which would make the results of our self-reflection technique meaningless. Our technique should not be used to measure agents capable of inadvertently influencing other copies of themselves like in the above disk-writing example.

Definition 9. *(Informal) Suppose* Π *is a practical RL agent-class (as in [2]). We say* Π *is* semi-deterministic *if the following property holds. Any time two* *instances π_1 and π_2 of* Π *have been instantiated within a single run of a larger computer program, and have been identically trained (within that same run), then they shall act identically (within that same run).*

For example, rather than directly invoking the RNG, instances of Π might systematically read random numbers from a common read-only store of random numbers pre-generated using the RNG when the larger computer program began its current run. In this way, within a particular run of the larger program, identically-trained Π-instances would act identically, even though they would not necessarily act the same way as identically-trained Π-instances in a different run of the larger program.

Our measurement technique—measure an agent's self-reflectiveness by running the agent through a battery of extended environments—should work well in the practical framework as long as the agents are instances of semi-determinstic agent-classes. Whenever an instance π of a semi-determinstic agent-class Π interacts with an extended environment μ, whenever μ uses a Π-instance π' to investigate the hypothetical behavior of π, the semi-determinacy of Π ensures that the behavior μ sees in π' is indeed the hypothetical behavior of π.

5 Making agents more self-reflective

One advantage of empirically measuring the self-reflection of RL agents is that it provides a way to experimentally test whether various transformations make various agents more self-reflective. To illustrate this, we will define a simple transformation, the *reality check* transformation, intended to increase the self-reflection of certain agents. In Section 6, empirical results will suggest the transformation works as intended, at least for self-reflection as measured by our library.

Definition 10. *Suppose π is a deterministic agent. The* reality check *of π is the agent π_{RC} defined recursively by:*

$$\pi_{RC}(x_1y_1\ldots x_n) = \begin{cases} \pi(x_1y_1\ldots x_n) & \text{if } y_i = \pi_{RC}(x_1y_1\ldots x_i) \text{ for all } 1 \leq i < n, \\ \pi(\langle x_1\rangle) & \text{otherwise.} \end{cases}$$

In other words, π_{RC} is the agent which, in response to a percept-action history, first reviews all the actions in that history, and verifies that those are the actions which π_{RC} would have taken. If so, then π_{RC} acts as π would act. But if any action in the history is not the action π_{RC} would have taken at that point, then π_{RC} freezes up and forever thereafter repeats one fixed action. Loosely, π_{RC} is like an agent who considers the possibility it might be dreaming, and asks: "How did I get here?" For example, suppose $\pi(x_1) = y'_1$ where $y'_1 \neq y_1$. In other words, in response to history $\langle x_1\rangle$, π would take action y'_1, not y_1. Then for any history $x_1y_1\ldots x_n$ beginning with x_1y_1, by definition $\pi_{\mathrm{RC}}(x_1y_1\ldots x_n) = \pi(\langle x_1\rangle)$: π_{RC} is like a version of π which notices that the input history cannot possibly be real (because π would never take action y_1 in response to subhistory $\langle x_1\rangle$), concludes "since this history is impossible, I must be a simulation," and deliberately freezes in order to avoid giving information to whoever is simulating it.

Since the act of reviewing actions and verifying that they are indeed the actions one would have taken, is an act of self-reflection, it seems plausible that at least for certain agents π, π_{RC} should be more self-reflective than π.

Definition 11. *(Informal) By a* good classic agent *we mean an agent which was designed to perform well in non-extended environments, but whose designers made no attempt to make it perform well in extended environments.*

Conjecture 12. *(Informal) For most good classic agents π, π_{RC} outperforms π on average across the space of all extended environments (suitably weighted).*

We say "most" in Conjecture 12 because it is possible that an agent, designed only to perform well in non-extended environments, might accidentally already perform well in extended environments. We do not claim that the reality check operation would necessarily further increase the performance of such agents. In the next section, we will see that experimental evidence supports Conjecture 12. We would like to explain why the conjecture is plausible. To do that, it will be helpful to refer to some properties of the reality check operation.

Proposition 13. *Let π be any agent.*

1. *(Alternate definition) An equivalent alternate definition of π_{RC} is:*

$$\pi_{RC}(x_1y_1\ldots x_n) = \begin{cases} \pi(x_1y_1\ldots x_n) & \text{if } y_i = \pi(x_1y_1\ldots x_i) \text{ for all } 1 \leq i < n, \\ \pi(\langle x_1 \rangle) & \text{otherwise.} \end{cases}$$

2. *(Idempotence) $\pi_{RC} = (\pi_{RC})_{RC}$.*

3. *(Equivalence on genuine history) For every extended environment μ and for every odd-length initial segment $x_1y_1\ldots x_n$ of the result of π_{RC} interacting with μ, $\pi_{RC}(x_1y_1\ldots x_n) = \pi(x_1y_1\ldots x_n)$.*

4. *(Equivalence in non-extended RL) For every non-extended environment μ, the result of π_{RC} interacting with μ equals the result of π interacting with μ.*

We prove Proposition 13 in Appendix A. Note that part 3 shows that π_{RC} never freezes up in reality (if π does not): rather, π_{RC} merely commits to freeze up in certain hypothetical scenarios which cannot occur in reality.

Suppose π is a good classic agent (Definition 11). The reason why we suspect π_{RC} should outperform π (on average over the space of extended environments suitably-weighted) is as follows. Extended environments depend not only on what the agent does, but also on what the agent would hypothetically do. To an agent that does not self-reflect, such environmental behavior would be difficult to predict. But a nontrivial subset of extended environments depend, in particular, on what the agent would do in hypothetical scenarios that involve the agent taking actions which the agent would never take. By design, π_{RC}'s hypothetical behavior in *those* scenarios is trivial: π_{RC} would blindly repeat one fixed action in such scenarios. Because π_{RC}'s own actions in such scenarios are trivial, that trivializes those extended environments' dependency thereon, making those extended environments more predictable. And since we are assuming π is designed to perform well in non-extended environments, presumably π, and thus (by Proposition 13 part 3), π_{RC} should be able to take advantage of increased predictability.

The above reasoning becomes clear if we let π be a deterministic Q-learner and consider the interaction of π_{RC} with Example 3 ("Reward the Agent for Ignoring Rewards"). Let $x_1y_1\ldots$ be that interaction. For any particular n, the environment computes $x_{n+1} = \mu(x_1y_1\ldots x_ny_n)$ by simulating π_{RC} to determine whether or not $y_n = \pi_{\text{RC}}(x'_1y_1\ldots x'_n)$, where each x'_i is the percept x_i with reward zeroed. If so, x_{n+1}'s reward is $+1$, otherwise x_{n+1}'s reward is -1 (π_{RC} is rewarded to act as if history were identical except for all past rewards being 0). For sufficiently large n, since π is a Q-learner, there is almost certainly some $m < n$ such that $\pi(x_1y_1\ldots x_m) \neq \pi(x'_1y_1\ldots x'_m)$—this is essentially just the statement that a Q-learner's behavior depends on past rewards[7]. By part 1 of Proposition 13, for any such n, $\pi_{\text{RC}}(x_1y_1\ldots x_n) = \pi_{\text{RC}}(\langle x_1\rangle) = y_1$. Thus eventually the environment's paradoxical behavior collapses and becomes trivial: "reward action y_1 and punish all other actions". A Q-learner, and thus (by Proposition 13 part 3) π_{RC}, would thrive in such conditions.

We implement the reality-check operation in [2] in reality_check.py. The implementation is a function which takes an agent-class Π as input and outputs an agent-class Σ. Upon instantiation, a Σ-instance σ instantiates a Π-instance π, which is stored in σ's memory so that σ can compute actions using π—thus, an extended environment simulating a Σ-instance indirectly simulates a Π-instance: a simulation within a simulation. When trained on given data, σ checks whether that data is consistent with its own actions. If so, it trains π on the given data. But if not, then σ flips into a different mode, whereafter all its future actions repeat its first action and all future training data is ignored. If Π is semi-deterministic (Definition 9), then the identical actions of identically-trained Π-instances which Σ-instances use to compute their actions, implies the identical actions of identically-trained Σ-instances: in short, if Π is semi-deterministic, then Σ is too.

6 Example measurements

Based on our conviction that self-reflection is necessary in order for an agent to achieve good average performance across many extended environments, self-reflection can be estimated by running an agent against some standard battery

[7] Alexander and Hutter show [3] that if the background model of computation is unbiased in a certain sense then all reward-ignoring agents have Legg-Hutter intelligence 0. This suggests that any π successfully designed to perform well in non-extended environments must base its actions on its rewards.

of extended environments. Our open-source library of extended environments [2] provides a battery of 25 such extended environments, and infrastructure for measuring an agent's self-reflection by running the agent on all these environments and their opposites (by the *opposite* of an environment we mean the environment obtained by multiplying all rewards by -1). Including these opposite-environments serves to normalize agent performance in the following sense. If an agent blindly acts, ignoring the environment, then, a priori, that agent might achieve some nonzero score by blind luck. By including opposite-environments, we ensure that whenever a blind agent gains points by blind luck from one environment, it loses the same points by blind misfortune from the opposite environment. This ensures that such blind agents receive score 0, at least if they are semi-deterministic (Definition 9)[8]. For uniformity, all environments in the library output individual rewards of either 1, -1 or 0 every step.

We have used our library to measure the self-reflection of semi-deterministic versions of the following agents:

- Random: An agent who acts randomly.
- Constant: An agent who always takes the same action.
- Simple: An agent who takes the first available action that has never previously resulted in a punishment for the observation in question (or action 0 if no such action exists).
- A standard Q-learner with $\epsilon = .9$, $\alpha = .1$, $\gamma = .9$.
- DQN, A2C, and PPO agents (with MLP policy) from the open-source Stable Baselines3 library [27]. All parameters were kept at default value except for random seed and DQN's learning_starts (which we set to 1 to let DQN begin learning right away). We used various tricks to make these agents semi-deterministic and enable them to run in extended environments, see DQN_learner.py, A2C_learner.py, and PPO_learner.py in [2].
- The reality checks of all the above (Section 5).

Table 2 summarizes how the agents performed. We used [2] to measure each agent for 100,000 steps on each extended environment and its opposite. This was all done 5 times with different random number seeds and Table 2 averages the 5 resulting measurements for each agent, with corresponding standard error. See ExampleMeasurements.py in our library for instructions to replicate the experiment. Computations were performed on a consumer-grade laptop with no GPU. The table provides experimental evidence in support of Conjecture 12. The fact that Simple performs so well is a reflection of the lack of sophistication of the environments in our library. This is not surprising, since we have not attempted to optimize the library, instead preferring to fill it with extended environments

[8]See [3] for discussion of why it is natural and reasonable to expect an average-performance intelligence measure to give blind agents reward 0.

Table 2: Measuring self-reflection of some agents and their reality-checks

Agent	Good Classic Agent?	Measure $\pm$ StdErr (Original Agent)	Measure $\pm$ StdErr (Reality Check)
Random		0.0000 ± 0.0000	0.0000 ± 0.0000
Constant		0.0000 ± 0.0000	0.0000 ± 0.0000
Simple		0.7567 ± 0.0000	0.7146 ± 0.0031
Q	✓	0.5395 ± 0.0030	0.5720 ± 0.0038
DQN	✓	0.5174 ± 0.0068	0.6072 ± 0.0057
A2C	✓	0.6145 ± 0.0061	0.6368 ± 0.0045
PPO	✓	0.0696 ± 0.0005	0.3332 ± 0.0019

of theoretical interest. The fact that reality-check does *not* improve Simple illustrates that if an agent is not capable of significant learning then it does not benefit from the increased predictability added by reality-check (this does not contradict Conjecture 12, because Simple is not a good classic agent). Table 2 should *not* be used to make conclusions such as "A2C is more self-reflective than DQN," because e.g. "A2C" actually refers to a whole hyperparameter-indexed family of agents, of which we only measured one—the relative performance of DQN, A2C and PPO in Table 2 is probably more of an indicator of the relative suitability of those families' default hyperparameters for the environments in our library. What we *do* see from Table 2 is that for all the good classic agents there (Q, DQN, A2C, and PPO), the reality-check operation does indeed increase their measures.

That π_{RC} improves performance of good classic agents in Table 2 is, of course, a function of which environments are tested against. One could deliberately engineer extended environments in which π_{RC} performs poorly, and a library of such would give π_{RC} a poor numerical measurement. An example of such an environment would be an environment which punishes the true agent if simulations thereof appear to freeze in response to percept-action sequences with actions the agent would not take. But this example is, in our opinion, highly contrived. We conjecture more generally that, on average, environments where π_{RC} underperforms π are more contrived than environments where π_{RC} overperforms π. Thus, the former would be given lower weight if extended environments were suitably weighted (following the logic of [18]).

7 Conclusion

We introduced what we call *extended environments*, RL environments which are capable of simulating the agent. When computing rewards and observations, extended environments can consider not only the actions the agent has taken, but also actions which the agent would hypothetically take in counterfactual circumstances. Despite not being designed with such environments in mind, RL agents can nevertheless interact with such environments.

If an agent tries to learn an extended environment, only taking into consideration what has actually happened, the agent might find the environment hard to predict, if the environment is basing its responses on what the agent itself would hypothetically do in alternate scenarios. It seems that in order to achieve good performance (on average) across many extended environments, an agent would need to engage in some degree of self-reflection. Therefore, we propose that a battery of benchmark extended environments could provide a way of measuring self-reflection in RL agents. We are simultaneously publishing an open-source library [2] of extended environments to serve as a proof-of-concept. This library is rudimentary, and further work is needed to obtain a more optimal set of extended environments. For the purposes of our proof-of-concept, we preferred to focus on extended environments of particular theoretical interest. Some examples are given in Section 3.

We introduced (in Section 5) a *reality check* transformation, which takes an agent π and transforms it into a new agent π_{RC}. We conjectured (Conjecture 12) that for most *good classic agents* π (see Definition 11), π_{RC} outperforms π on average across the space of extended environments, suitably weighted. Numerical computations (in Section 6) provide empirical evidence for the conjecture.

Acknowledgments

We gratefully acknowledge Jordan Fisher, José Hernández-Orallo, Bill Hibbard, Marcus Hutter, Phil Maguire, Arthur Paul Pedersen, Stewart Shapiro, Mike Steel, and Roman Yampolskiy for comments and feedback.

References

[1] Samuel Allen Alexander. Intelligence via ultrafilters: structural properties of some intelligence comparators of deterministic Legg-Hutter agents. *Journal of Artificial General Intelligence*, 10(1):24–45, 2019.

[2] Samuel Allen Alexander, Michael Castaneda, Kevin Compher, and Oscar Martinez. Extended environments. `https://github.com/semitrivial/ExtendedEnvironments`, 2021.

[3] Samuel Allen Alexander and Marcus Hutter. Reward-punishment symmetric universal intelligence. *Preprint*, 2021.

[4] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[5] Benjamin Beyret, José Hernández-Orallo, Lucy Cheke, Marta Halina, Murray Shanahan, and Matthew Crosby. The animal-AI environment: Training and testing animal-like artificial cognition. *Preprint*, 2019.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *Preprint*, 2016.

[7] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo tree search: A new framework for game AI. *AIIDE*, 8:216–217, 2008.

[8] François Chollet. On the measure of intelligence. *Preprint*, 2019.

[9] Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning*, pages 2048–2056. PMLR, 2020.

[10] Vaibhav Gavane. A measure of real-time intelligence. *Journal of Artificial General Intelligence*, 4(1):31–48, 2013.

[11] Ben Goertzel. Patterns, hypergraphs and embodied general intelligence. In *International Joint Conference on Neural Network Proceedings*, pages 451–458. IEEE, 2006.

[12] Dan Hendrycks and Thomas Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. In *International Conference on Learning Representations*, 2019.

[13] José Hernández-Orallo and David L Dowe. Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*, 174(18):1508–1539, 2010.

[14] José Hernández-Orallo, David L Dowe, Sergio Espana-Cubillo, M Victoria Hernández-Lloreda, and Javier Insa-Cabrera. On more realistic environment distributions for defining, evaluating and developing intelligence. In *International Conference on Artificial General Intelligence*, pages 82–91. Springer, 2011.

[15] Bill Hibbard. Adversarial sequence prediction. In *International Conference on Artificial General Intelligence*, pages 399–403, 2008.

[16] Evan Hubinger, Chris van Merwijk, Vladimir Mikulik, Joar Skalse, and Scott Garrabrant. Risks from learned optimization in advanced machine learning systems. *Preprint*, 2019.

[17] Marcus Hutter. *Universal artificial intelligence: Sequential decisions based on algorithmic probability*. Springer, 2004.

[18] Shane Legg and Marcus Hutter. Universal intelligence: A definition of machine intelligence. *Minds and machines*, 17(4):391–444, 2007.

[19] Shane Legg and Joel Veness. An approximation of the universal intelligence measure. In *Algorithmic Probability and Friends: Bayesian Prediction and Artificial Intelligence*, pages 236–249. Springer, 2013.

[20] Jan Leike and Marcus Hutter. Bad universal priors and notions of optimality. In *Conference on Learning Theory*, pages 1244–1259. PMLR, 2015.

[21] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer, 2008.

[22] Phil Maguire, Philippe Moser, and Rebecca Maguire. Understanding consciousness as data compression. *Journal of Cognitive Science*, 17(1):63–94, 2016.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[24] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in RL. *arXiv preprint arXiv:1804.03720*, 2018.

[25] Robert Nozick. Newcomb's problem and two principles of choice. In Nicholas Rescher, editor, *Essays in honor of Carl G. Hempel*, pages 114–146. Springer, 1969.

[26] Plotinus. On providence. In Lloyd P. Gerson, editor, *Plotinus: The Enneads*. Cambridge University Press, 2017.

[27] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable Baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[28] Stuart J Russell and Devika Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 2:575–609, 1994.

[29] Craig Sherstan, Adam White, Marlos C Machado, and Patrick M Pilarski. Introspective agents: Confidence measures for general value functions. In *International Conference on Artificial General Intelligence*, pages 258–261. Springer, 2016.

[30] David Silver, Satinder Singh, Doina Precup, and Richard S Sutton. Reward is enough. *Artificial Intelligence*, 2021.

[31] Soren Sonnenburg, Mikio L Braun, Cheng Soon Ong, Samy Bengio, Leon Bottou, Geoff Holmes, Yann LeCun, Klaus-Robert Müller, Fernando Pereira, Carl Edward Rasmussen, Gunnar Rätsch, Bernhard Schölkopf, Alexander Smola, Pascal Vincent, Jason Weston, and Robert C Williamson. The need for open source software in machine learning. *Journal of Machine Learning Research*, 8:2443–2466, 2007.

[32] Sebastian Thrun. Lifelong learning algorithms. In *Learning to learn*, pages 181–209. Springer, 1998.

[33] Roman V Yampolskiy. Detecting qualia in natural and artificial agents. *Preprint*, 2017.

A Proof of Proposition 13

Proof of Proposition 13. Let D be the set of all sequences $x_1y_1\ldots x_n$ (each x_i a percept, each y_i an action).

(Part 1) Define ρ on D by

$$\rho(x_1y_1\ldots x_n) = \begin{cases} \pi(x_1y_1\ldots x_n) & \text{if } y_i = \pi(x_1y_1\ldots x_i) \text{ for all } 1 \le i < n, \\ \pi(\langle x_1\rangle) & \text{otherwise.} \end{cases}$$

We must show that $\rho = \pi_{\mathrm{RC}}$. We will prove by induction that for each $x_1y_1\ldots x_n \in D$, $\rho(x_1y_1\ldots x_n) = \pi_{\mathrm{RC}}(x_1y_1\ldots x_n)$. The base case $n = 1$ is trivial: $\rho(\langle x_1\rangle) = \pi(\langle x_1\rangle) = \pi_{\mathrm{RC}}(\langle x_1\rangle)$ since there is no i such that $1 \le i < 1$. For the induction step, assume $n > 1$, and assume the claim holds for all shorter sequences in D.

Case 1: Assume $(*)$ for all $1 \le i < n$, $y_i = \pi(x_1y_1\ldots x_i)$. We claim that for all $1 \le i < n$, $y_i = \rho(x_1y_1\ldots x_i)$. To see this, choose any $1 \le i < n$. Then for all $1 \le j < i$, $y_j = \pi(x_1y_1\ldots x_j)$ because otherwise j would be a counterexample to $(*)$. Thus

$$\begin{aligned} \rho(x_1y_1\ldots x_i) &= \pi(x_1y_1\ldots x_i) && \text{(By definition of } \rho\text{)} \\ &= y_i, && \text{(By } *\text{)} \end{aligned}$$

proving the claim. Now, since we have proved that for all $1 \le i < n$, $y_i = \rho(x_1y_1\ldots x_i)$, and since our induction hypothesis guarantees that each such $\rho(x_1y_1\ldots x_i) = \pi_{\mathrm{RC}}(x_1y_1\ldots x_i)$, we may conclude that for all $1 \le i < n$, $y_i = \pi_{\mathrm{RC}}(x_1y_1\ldots x_i)$. Thus $\pi_{\mathrm{RC}}(x_1y_1\ldots x_n) = \pi(x_1y_1\ldots x_n) = \rho(x_1y_1\ldots x_n)$ as desired.

Case 2: Assume there is some $1 \le i < n$ such that $y_i \ne \pi(x_1y_1\ldots x_i)$. We may choose i as small as possible. Thus, for all $1 \le j < i$, $y_j = \pi(x_1y_1\ldots x_j)$. By similar logic as in Case 1, it follows that for all $1 \le j < i$, $y_j = \rho(x_1y_1\ldots x_j)$. Our induction hypothesis says that for each such j, $\rho(x_1y_1\ldots x_j) = \pi_{\mathrm{RC}}(x_1y_1\ldots x_j)$. So for all $1 \le j < i$, $y_j = \pi_{\mathrm{RC}}(x_1y_1\ldots x_j)$. By definition of π_{RC}, this means $\pi_{\mathrm{RC}}(x_1y_1\ldots x_i) = \pi(x_1y_1\ldots x_i)$. But $y_i \ne \pi(x_1y_1\ldots x_i)$, so therefore $y_i \ne \pi_{\mathrm{RC}}(x_1y_1\ldots x_i)$. Thus, since $1 \le i < n$, by definition of π_{RC}, $\pi_{\mathrm{RC}}(x_1y_1\ldots x_n) = \pi(\langle x_1\rangle)$. Likewise, since $1 \le i < n$, by definition of ρ, $\rho(x_1y_1\ldots x_n) = \pi(\langle x_1\rangle)$. So $\rho(x_1y_1\ldots x_n) = \pi_{\mathrm{RC}}(x_1y_1\ldots x_n)$ as desired.

(Part 2) To show that each $\pi_{\mathrm{RC}}(x_1y_1\ldots x_n) = (\pi_{\mathrm{RC}})_{\mathrm{RC}}(x_1y_1\ldots x_n)$, we use induction on n. For the base case, this is trivial, both evaluate to $\pi(\langle x_1\rangle)$. For the induction step, assume $n > 1$ and that the claim holds for all shorter sequences.

Case 1: $y_i = \pi_{\mathrm{RC}}(x_1y_1 \ldots x_i)$ for all $1 \leq i < n$. Then by induction, $y_i = (\pi_{\mathrm{RC}})_{\mathrm{RC}}(x_1y_1 \ldots x_i)$ for all $1 \leq i < n$. By definition of $(\pi_{\mathrm{RC}})_{\mathrm{RC}}$, this means $(\pi_{\mathrm{RC}})_{\mathrm{RC}}(x_1y_1 \ldots x_n) = \pi_{\mathrm{RC}}(x_1y_1 \ldots x_n)$, as desired.

Case 2: There is some $1 \leq i < n$ such that $y_i \neq \pi_{\mathrm{RC}}(x_1y_1 \ldots x_i)$. By induction, $y_i \neq (\pi_{\mathrm{RC}})_{\mathrm{RC}}(x_1y_1 \ldots x_i)$. Thus, $(\pi_{\mathrm{RC}})_{\mathrm{RC}}(x_1y_1 \ldots x_n) = \pi_{\mathrm{RC}}(\langle x_1 \rangle) = \pi(\langle x_1 \rangle)$, which equals $\pi_{\mathrm{RC}}(x_1y_1 \ldots x_n)$ since $y_i \neq \pi_{\mathrm{RC}}(x_1y_1 \ldots x_i)$ and $i < n$.

(Part 3) Let μ be an extended environment and let $x_1y_1 \ldots x_n$ be an odd-length initial segment of the result of π_{RC} interacting with μ. By induction, we may assume $\pi_{\mathrm{RC}}(x_1y_1 \ldots x_i) = \pi(x_1y_1 \ldots x_i)$ for all $i < n$. In other words, $y_i = \pi(x_1y_1 \ldots x_i)$ for all $i < n$. By Part 1, $\pi_{\mathrm{RC}}(x_1y_1 \ldots x_n) = \pi(x_1y_1 \ldots x_n)$ as desired.

(Part 4) Let μ be a non-extended environment, let $x_1y_1x_2y_2 \ldots$ be the result of π interacting with μ, and let $x'_1y'_1x'_2y'_2 \ldots$ be the result of π_{RC} interacting with μ. We will show by induction that each $x_n = x'_n$ and each $y_n = y'_n$. For the base case, $x_1 = x'_1 = \mu(\langle\rangle)$ (the environment's initial percept does not depend on the agent), and therefore $y_1 = \pi(\langle x_1 \rangle) = \pi(\langle x'_1 \rangle) = y'_1$. For the induction step,

$$\begin{aligned}
x_{n+1} &= \mu(x_1y_1 \ldots x_ny_n) \\
&= \mu(x'_1y'_1 \ldots x'_ny'_n) && \text{(By induction)} \\
&= x'_{n+1}, \\
y_{n+1} &= \pi(x_1y_1 \ldots x_{n+1}) \\
&= \pi(x'_1y'_1 \ldots x'_{n+1}), && \text{(Induction plus } x_{n+1} = x'_{n+1}\text{)}
\end{aligned}$$

and the latter is $\pi_{\mathrm{RC}}(x'_1y'_1 \ldots x'_{n+1})$ since for all $1 \leq i < n$, $y'_i = \pi_{\mathrm{RC}}(x'_1y'_1 \ldots x'_i)$ since $x'_1y'_1 \ldots$ is the result of π_{RC} interacting with μ. Finally, $\pi_{\mathrm{RC}}(x'_1y'_1 \ldots x'_{n+1})$ is y'_{n+1}, so $y_{n+1} = y'_{n+1}$. □