2021

Markov Decision Processes with Embedded Agents

Luke Harold Miles
University of Kentucky, luke@cs.uky.edu
Digital Object Identifier: https://doi.org/10.13023/etd.2021.135

Luke Harold Miles, Student

Dr. Brent Harrison, Major Professor

Dr. Zongming Fei, Director of Graduate Studies

Markov Decision Processes with Embedded Agents

THESIS

A thesis submitted in partial
fulfillment of the requirements for
the degree of Master of Science in
the College of Engineering at the
University of Kentucky

By
Luke Harold Miles
Lexington, Kentucky

Director: Dr. Brent Harrison, Professor of Computer Science
Lexington, Kentucky 2021

ABSTRACT OF THESIS

Markov Decision Processes with Embedded Agents

We present Markov Decision Processes with Embedded Agents (MDPEAs), an extension of multi-agent POMDPs that allow for the modeling of environments that can change the actuators, sensors, and learning function of the agent, e.g., a household robot which could gain and lose hardware from its frame, or a sovereign software agent which could encounter viruses on computers that modify its code. We show several toy problems for which standard reinforcement-learning methods fail to converge, and give an algorithm, ‘just-copy-it‘, which learns some of them. Unlike MDPs, MDPEAs are closed systems and hence their evolution over time can be treated as a Markov chain. In future work, we hope MDPEAs can be extended to model even fully embedded agents acting in real digital or physical environments.

KEYWORDS: MDP, Reinforcement Learning, Embedded Agency

Author's signature: Luke Harold Miles

Date: May 13, 2021

Markov Decision Processes with Embedded Agents

By
Luke Harold Miles

Director of Thesis: Brent Harrison

Director of Graduate Studies: Zongming Fei

Date: May 13, 2021

ACKNOWLEDGMENTS

Dr. Harrison is a remarkably patient and persistent advisor, and I could not have begun to do this work without his thoughtful guidance. Dr. Judy Goldsmith made me feel at home in the CS department and always lent me her piercing intellectual eye when I needed it. Dr. Raphael Finkel encouraged me to pursue research and follow my interests and present them to the department. My ex-girlfriend Luci Keller encouraged me to pursue a Master's degree and sat me through countless nights of code refactoring. My mother Colleen London and father Hal Miles gave me helpful reminders to finish, with the confidence that I could. My girlfriend Asya Kudriashova supported me through the last few months. I owe all of you a great debt.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 Introduction

Sequential decision problems are problems where an agent must act and plan in an environment where its actions have long-term consequences. Sequential problems are different from e.g., classification problems because in the latter there is no notion of planning or consequence. Sequential problems can be complex or simple, discrete or continuous, supervised or unsupervised, but in all cases, an agent is acting in an environment over time to achieve a goal or maximize a utility or reward function.

One important formalization of sequential problems is Markov decision processes (MDPs), which break it into states, actions, a transition function, and a reward function. MDPs are an extremely expressive model; they can represent anything from board games and video games to resource management and process scheduling to stock trading to autonomous vehicles to theorem proving.

An MDP can be considered as a precise statement of a sequential problem; its solution is a *policy* that maps each state to the action which maximizes expected reward.[1] Reinforcement learning (RL) is the study of algorithms that take an MDP as input and produce a policy for that environment as output. Most RL methods rely on trial and error in an environment to learn its dynamics and create a good policy: take semi-random action, record the results, and use that information.

Reinforcement learning is only fully applicable[2] to situations where

(1) the agent cannot be destroyed or killed,
(2) the agent's actuators cannot be modified, added to, or removed,
(3) the agent's sensors cannot be modified, added to, or removed,
(4) the agent's learning algorithm and parameters cannot be changed by and are not observable to other agents,
(5) the agent's parameter space is large enough to model the environment extensively,
(6) the environment provides an incorruptible, invisible, well-specified reward channel directly to the agent, and
(7) no single action will permanently lockout more than a 'small amount' of utility[3].

These assumptions fail in core potential application areas of RL. For example, in robotics, which is at the surface well-suited to be modeled as an MDP, in practice, most training can only be done in simulation, or else the robot would risk damaging itself or destroying things or hurting people [1,2]. Simulations are always lower fidelity than the real world, so many tasks cannot really be simulated at all, or else the robot trained only in simulation has unexpected failures once deployed. A simple example of

[1]Policies can be probabilistic; in that case they map each state to an optimal distribution over next-actions.

[2]'fully applicable' meaning that e.g., simulations will be accurate, proofs of algorithms' correctness or convergence are valid, etc.

[3]sub-linear regret is a common definition of this notion

the difficulty of simulation: even modern physics engines essentially fail to represent what happens when one surface of one material touches another surface of another material [3]. These errors at the detailed level lead to macro-errors in the overall dynamics of a system.

A variety of scenarios and ways that the assumption of a cleanly-split and catastrophe-free environment can fail is demonstrated by Leike et al. [4]. Agents may be able to gain access to the reward channel, permanently lock out all reward with a damaging action, or exploit a misspecified reward function to 'solve' the MDP without completing an intended task.

What is needed here? How should these challenges be addressed? We contend that small patches to augment learning algorithms or MDPs for specific use-cases are insufficient and that a significant reworking of the model is needed. For example, human intervention to prevent catastrophes [5] cannot be used when actions must be taken on the scale of milliseconds, as in the case of e.g. stock-trading algorithms, which have more examples of catastrophic failures than can be enumerated (e.g. the 2010 flash crash [6]).

What is the reality of sequential problems? Let us start from scratch and see what model seems natural. First, agents are never outside of the environment, never completely separated from it. All sandboxed environments, such as controller input in Super Mario World [7], embedded javascript in messages in Microsoft Teams [8], and even air-gapped computers in Faraday cages [9,10] have some kind of memory leak or other way the barrier breaks down. So we must begin with the notion that the agent and the environment are tangled.

Secondly, no environment is truly episodic. There is no 'reset level' button or timer in real environments. In fact, the *most important characterizing property* of sequential problems may be **permanent consequences**. This is partially captured in the notion of single-/few-shot learning [11] or lifelong learning [12], but we should make it central to our model.

One could go so far as to say the notion of agents is itself confused, and that an environment is simply matter proceeding through time according to the laws of physics. We are not so extreme; agents seem like a very real and natural concept, and dissolving them leaves the model with little to offer, and renders it incompatible with the extensive tooling developed for reinforcement learning. As a compromise between tractability and accuracy, we make the following assumptions:

(1) There is an environment which is entirely closed excepting the reward and transition functions.
(2) There are agents in this environment, although they may be created, modified, and destroyed.
(3) Agents perceive the environment and each other through sensors, although which sensors a particular agent possesses may change.[4]
(4) Agents act on the environment through actuators, although agents may gain or lose them.

[4]Sensors, reward functions, and the transition function can all be made probabilistic.

(5) Each agent stores memories, algorithms, etc., in some kind of body or brain that is itself part of the environment, although it is subject to harm or improvement by the agent itself, by the environment, or by other agents.
(6) Each agent is entirely defined by its sensors, actuators, and brain.
(7) There is a transition function which specifies how the environment changes over time and what each actuator does.
(8) Hence there must also be an environment-wide clock[5] and each time it ticks, all agents take actions and the entire environment updates simultaneously, according to the transition function.
(9) When an agent is created, the transition function assigns it a reward function, which gives the brain of the agent a real-numbered reward at each timestep, although the agent may ignore this value.[6]

We call this model a Markov decision process with embedded agents (MDPEA). Our hope is that simulations of MDPEAs are **more accurate** to the real problems being modeled (because they can represent a broader category of dynamics in the world) and that this leads to learning algorithms which are both **more reliable** (because they can detect and avoid possible dangers that would otherwise be invisible), and **more performant** (because they can make changes and do handoffs that would otherwise be unavailable). Aside from the perspective and model itself, we make two contributions.

First, we present a test suite `MDPEA-gym` of three code-implemented environments and three described environments which give different learning challenges for agents and reveal different limitations of MDPs. These address issues from multi-agent trust when recursive inspection is possible to using an environment to improve an algorithm to avoiding permanent loss.

Second, we describe an algorithm, `just-copy-it`, that acts in some of these environments and successfully evaluates self-modifying actions, taking beneficial ones and avoiding harmful ones. This task is relatively narrow and our algorithm only solves it in ideal conditions; much room is left for future learning algorithms that solve other challenges in MDPEAs.

[5]Or an outside observer can imagine such a clock without loss of information

[6]We keep reward functions for compatibility, but they can be discarded for more reasonable agent-originating utility functions when possible.

Chapter 2 Background

A *Markov decision process* (MDP) models a single agent acting in an environment and is defined as a tuple $M = \langle S, A, T, R, \gamma \rangle$.

- S is the set of possible world states that the agent could possibly be in. This could be the position of the agent in a maze, the velocity and position of balls on a billiards table, or any other fully-observable and Markovian state that does not include the agent or policy. The set of states can be finite or infinite, and discrete or continuous.
- A is the set of actions that the agent can take. Typically, the agent is permitted to take any action in any state, but if an action does not 'make sense,' then its effect is null. It may be infinite or continuous. Examples of actions in a variety of domains: making a buy order in a stock-trading environment; going a direction in a maze; applying a particular angle, force, and position of a pool cue in billiards.
- $T : S \times A \to S$ is the transition function mapping each state and action to the next state. It may have probabilistic output. In the billiards example, the physics of the bouncing balls is encoded in the transition function.
- $R : S \times A \to S$ is the reward function, and maps each state-action pair to a particular 'reward,' meant to indicate success on the task. Note that this comes from the environment and is itself not part of the agent. The agent tries to take those actions which maximize cumulative discount reward.
- $0 < \gamma \leq 1$ is the discount factor, which signifies how much more valuable reward is sooner than later..[1]]

Note that the policy itself, and the learning algorithm creating the policy, are not considered part of M. Reward R and discount γ are not even used by a policy, only by the learning algorithm and for evaluation, so they can be discarded when running a fixed policy in an environment.

MDPs are a mathematical construction, but are subject to some constraints when used in a computational setting, such as time-complexity analysis or execution on a computer. For example, the transition function must be computable, each state and action must be finitely representable, and the sets of actions and states must typically be computationally enumerable. This rules out, for example, using arbitrary real numbers for reward instead of floating-points.

Reinforcement learning [15] is a collection of methods for solving MDPs. A solution to an MDP is an optimal policy, i.e. a function $\pi : S \to A$ which (probabilistically) takes that action that maximizes the expected cumulative reward in the

[1]One problematic aspect of discount factors is that, at some time horizon, the destruction of the universe costs less than avoiding eating a single Cheeto now. This is rarely a problem in practice. See discussion here [13,14]

environment, given that the agent will continue to follow the policy. I.e.,

$$\pi^* := \arg\max_{\pi} \mathbb{E}[\pi], \text{where}$$

$$\mathbb{E}[\pi] := \mathbb{E}\left[\sum_{t=t_0}^{\infty} R(S_t, A_t)\gamma^t \middle| \pi\right].$$

Two extremely simple policies are the random policy and the constant policy, and they serve as simple benchmarks to compare against. An elegant and versatile algorithm for creating policies is tabular Q-learning [16], which stores a table of estimated expected values of each state-action pair.[2] We will use Q-learning as the base of `just-copy-it`.

A *stochastic game* [17] is essentially the multi-agent version of an MDP. It is defined as a tuple $\langle S, A_1, \ldots, A_n, T, R_1, \ldots, R_n \rangle$ where n is the number of agents. All agents share the same environment and transition function, but each agent has its own action set and reward function. A stochastic game is considered to be fully observable in the sense that the entire state is visible to every agent, but this is misleading: from the perspective of an individual agent, the environment is no longer Markovian. In this sense, all stochastic games are POMDPs. It is conceivable, e.g. in robotics, that one agent could perceive the available actions or the reward function of another agent. For that matter, even the internal learning algorithm of another agent is sometimes perceptible. We aim to model this.

A *partially-observable Markov decision process* (POMDP) is a 7-tuple $\langle S, A, T, R, \Omega, O, \gamma \rangle$, where

- S, A, T, R, γ are as in an MDP,
- Ω is the set of possible observations, and
- O is the set of conditional observation probabilities.

A learning algorithm (e.g. Q-learning) in this environment does not provide a policy that maps states to actions, but instead effectively implements a function from a history of observation-action pairs to actions. This is crucial: in an MDP, the original learning algorithm can be discarded once a policy is produced, but in a POMDP, the learning algorithm is shackled to the simulation. The performance of an agent depends on its ability to model those dynamics of the statespace and transition function which are important for predicting reward; at one extreme, the agent can re-establish a Markovian representation of the state and use the powerful techniques available for that; at the other extreme, either the observations give no useful information or the agent is completely unable to compress the history, and each timestep the action is no better than random.

Note that POMDPs do not have a notion of sensors, but instead the environment directly delivers perceptions (observations) to the learning algorithm. However, sensors can still be represented. For example, you can simulate an agent picking up

[2] Its primary limitation is that it requires finite and discrete action and statespaces, but this can be overcome with a variety of methods including storing an approximated table as weights in a neural network.

a camera by transitioning to another 'layer' of the statespace where the transitions are the same, but O provides richer information to the learning algorithm. This is ontologically problematic: the agent has transitioned to a completely disjoint set of states, but none of the action dynamics seem to have changed.

MDPs, POMDPs, & stochastic games do not directly represent actuators either. To model picking up a hammer, essentially the environment transitions to another portion of statespace where the old actions have null effect and the new actions start to have an effect. But the new actions were not there before and the statespace should not really have changed! We aim to integrate sensors & actuators directly into our model to make it more philosophically consistent with the standard [18] actuator-sensor-brain model of agents.

Chapter 3 Related Work

This paper is largely inspired by [19], which pellucidly explains the challenges lying between today's RL framework and the goal of creating intelligent agents that are fully embedded in their environment. We place our framework, our proposed algorithm, and our test set in the context of the literature.

There have been countless efforts to extend the MDP to model a broader range of scenarios, from partial observability [20,21] and multiple agents [17,22] to parameterized actions spaces [23] to constraints on expected reward [24] to decentralized partial-observability [25]. The MDPEA is not strictly broader than all of these extensions in the sense that they are each precisely an instance of an MDPEA; however, the essential dynamics of each of them can be effectively *emulated* by an MDPEA. The goal of our framework is much broader and more ambitious than most other extensions: we aim to model *most* important aspects of multiple agents acting embedded in their environment.

MDPEAs are compatible with MDPs and standard RL algorithms, and we hope that they are directly applicable to useful, real-world problems. Other frameworks of embedded agents such as resource-bounded open-source game theory [26] or bounded-rational self-modifying agents [27] are in some ways more expressive, elegant, and theoretically tractable than MDPEAs, but are not immediately applicable.

Our algorithm, `just-copy-it` is a limited black-box approach for agents to evaluate self-modifying actions. This is a far more narrow goal than that of self-improving programs more generally and separate from the goals of minimizing side effects in the environment outside of the agent [5,28] or reducing the complexity of large action spaces [29].

The `MDPEA-gym` is meant to serve as a small standard test set that future learning algorithms can be evaluated on. This is in the same spirit as the OpenAI Gym [30] (a varied set of challenging reinforcement-learning test environments), [4] (a set of gridworlds meant to evaluate an agent's safety and robustness), and the [31] (which provides a suite of environments to demonstrate different properties of the universal Bayesian agent [32]). `MDPEA-gym` is aimed specifically at the MDPEA and notions of sensors and actions.

One work of note is the 1984 game called Core War [33], where two players each design an assembly program on a virtual machine, which are then executed in parallel until one program hijacks the other's instruction pointer. The game is a successful model of nearly fully-embedded agents[1] and due to its popularity has been subject to extensive analysis and has a large database of player programs, which are useful objects of study when designing other embedded algorithms. However, Core War is limited in that there is only ever one goal in the environment, and hence there is little for agents to learn, and it cannot represent common useful tasks.

[1]'Nearly' because the immutable 'instruction pointers' from the virtual machine is somewhat dualistic.

Our work is an effort to expand slightly out from the fully-dualistic framing of MDPs to a partially embedded one. Alternatively, one could start from a fully embedded model such as cellular automata or particle physics simulations, and try to build up notions of agents. Examples of such efforts are the formalization of preferences within physical world models [34], defining action and perception from the ground up [35], detecting emergent processes within a simulation at different granularities [36][2]. We choose not to make our model fully embedded to keep it partially compatible with existing learning algorithms and analysis techniques from the RL literature.

[2]e.g. gliders in Conway's game of life can be treated as a unit. Or in our world, cells can be treated as units for most purposes, planets can be treated as units for astrophysics, etc.

Chapter 4 Model

An *Markov decision process with embedded agents* $M = \langle S, A, T, R, P, Se, Br, \gamma \rangle$ is composed of

- a set S of states,
- a set A of actions,
- a transition function $T : S \times \langle Ag \rangle^* \times \langle A \rangle^* \to S \times \langle Ag \rangle^*$ which maps a state, a vector of agents, and their actions at time T to a state and vector of agents at time $T + 1$, where the number of agents may change,
  - (this enables the environment to add, remove, and modify the agents)
- a reward function $R : S \times \langle A \rangle^* \to \langle \mathbb{R} \rangle^*$ which maps each state and vector of actions to a vector of rewards,
  - (Alternatively, view the transition function as assigning a reward function to each agent when it is created.)
- a set P of perceptions,[1]
- a set Se of sensors, where each sensor is a function $se : S \times \langle \mathbb{R} \rangle^* \times \langle Ag \rangle^* \to P$ which maps the state of the environment, a vector of rewards, and a vector of agents to a perception,
- **a set** Br **of possible brains or learning algorithms**, where each $br : R \times 2^P \times 2^A \to A \times Br$ maps a reward, percept, and action set to a particular action, **and a new brain for time** $T + 1$,[2] and finally
- a discount factor γ.

One agent is defined a tuple $ag = \langle br, A_{ag}, Se_{ag} \rangle$ of a brain, actions, and sensors. The set Ag of agents above is the cross product of brains, action sets, and sensor tuples:

$$Ag := Br \times 2^A \times 2^{Se}.$$

Hence, the set of possible agents need not be specified in the definition of a particular MDPEA. However, it is sometimes helpful to have two more additional attributes:

- an initial state S_0, and
- an initial set of agents $\langle Ag_0 \rangle$.

The MDPEA is a closed process that proceeds through time without input, which makes it truly Markovian from an outside perspective. Simulation proceeds in the following order:

[1] We choose the term 'perception' instead of 'observation' as a reminder that the data comes from the sensors.

[2] This allows us to capture the change to the policy over time within the MDPEA itself.

```
procedure simulate(agents, state, transition, reward_of):
    # each agent is tuple of sensors, actions, and brain
    rewards = [0 for each agent]
    actions = empty array
    loop forever:
        simultaneously for each agent in agents:
            percept = [sensor(state, agents, rewards)
                      for each sensor in agent.sensors]
            action, agent.brain :=
                agent.brain(rewards[agent], percept, agent.actions)
            actions[agent] = action
        rewards = reward_of(state)
        state, agents := transition(state, agents, actions)
```

Note that there are no *episodes* strictly speaking in this model. An environment may teleport the agent back to a start state when it reaches a place, but in general, any effects of the agent's actions can persist indefinitely. Also note that, at each timestep, an agent first updates its own brain, then the environment may change the agent. Finally, states do not contain agents, but the transition function and all sensors have access to the vector of agents.

4.1 Consideration of framing

This model could be made more granular, e.g. by breaking the brain into `(procedure, parameters, memory)` or having distinct kinds of actions or sensors. It could also be made more coarse by collapsing the sensor set into a single perception function, or by eliminating the notion of perceptions and just having the agent brain act directly on the state. However, the present level of structure allows modifications to be characterized more easily, for example, some modifications only add sensors, which would not be as clear if the modification instead replaced an entire `sense` function. And when more detail is necessary on a class of objects, the set under consideration can be constrained and characterized, and transition defined in terms of that characterization, which we do in case study 1. In short, we've chosen the minimal level of detail necessary to capture what we believe to be the core problems in this area.

A second question is the computability of this framework. Sensors are functions mapping states, agents, and rewards to data, but each agent has its own set of sensors, and those, in turn, do take the first sensor as part of their input. This suggests that MDPEAs permit the description of environments which cannot actually be computed.[3] Even worse, it permits the description of environments that are mathematically impossible or otherwise nonsensical. Take this MDPEA for example:

- No actions, one state, two initial agents, and one sensor for each agent
- The first agent's sensor returns the output of the second agent's

[3]Or more reductively, a sensor could be defined to take a binary number lying in the environment and return whether or not a Turing machine executed on that program would halt.

- The second agent's sensor returns the output of the first agent's

One strength of MDPs (together with how policies are usually defined) is that their computational structure is acyclic, and therefore never has problems of recursive, undefined agent behavior or environment dynamics. However, when two humans, Alice and Bob, play rock-paper-scissors in the physical world, Alice does imagine what Bob might play and what he might imagine that Alice will play, etc., and this happens without either person melting or the physics breaking down. Humans do not have direct read-access to each other's minds[4], but we do have a sense of what algorithms each other are using, where they came from, how quickly they learn, and so on. We hold that such dual-recursive and agent-embedded processes are extremely common in the real world, will be even more common in the age of fully-readable digital agents, and that understanding them properly is essential to creating safe and effective learning agents in many domains.[5]

Ways of changing the model to exclude mathematically or computationally impossible environments without losing the capacity to model these kinds of recursive problems are discussed in the final section of this paper.

4.2 Proposed algorithm: `just-copy-it`

Recall that, in an MDPEA, an agent is defined as a tuple of `(brain, actions, sensors)` where the brain is a function taking `reward, perceptions, actions` and returning an action and updated brain function for the next timestep. The `just-copy-it` algorithm is a brain.

The core idea is that it proceeds normally through the environment as a standard RL agent, balancing exploration and exploitation, etc., until it finds a modifying state. Then, it creates a constrained copy of itself – one without copying capabilities which always avoids modifications – and observes the copy's reward for a fixed time period then deletes the copy. If the copy has a higher average reward than the original agent, then the algorithm makes the modification itself. This algorithm is intended specifically to solve the problem of taking positive modifications and avoiding harmful ones when they occur in the environment – finding optimal policies for MDPEAs in the general case is of course out of scope for this paper.[6]

[4]To the best of the authors' knowledge

[5]Such dynamics are not outright-forbidden by an MDP: one agent can use the history of actions of another agent to produce an estimate of its policy, and nothing stops a learning algorithm from taking into account that the other agents may have models of its own policy. A simple example of this: a rock-paper-scissors algorithm Alg might be programmed to start playing randomly and quit attempting to beat the Nash equilibrium if it loses ten times in a row. This behavior could be justified as "strategy mysteriously not working so try a different strategy" but the real justification for that programming is that Alg understands it has an inferior opponent model to its opponent. However, that reasoning cannot be expressed within the language of the MDP, which likely limits the capacity of agents using the MDP model to learn such strategies more generally, and limits the interpretability of agents, including their capacity to explain themselves .

[6]MDPEAs are more general than MDPs, POMDPs, or stochastic games and each of them have hundreds of papers about and years of attempts at methods for finding optimal policies.

For simplicity and consistency, the base learning algorithm here is Q-learning, but e.g. SARSA or n-step TD prediction could instead be substituted just as well. Additionally, for brevity, we present the procedure as modifying parameters in place, but this can easily be recast as one function returning a new updated function.

Short pseudocode[7]:

```
procedure just-copy-it(reward, perceptions, actions) → action:
    if just killed clone and the modification was good:
        return the action that the clone was initialized with
    else if clone is still being evaluated:
        record perceptions.clone_reward
        return actions.wait
    else if clone timer just finished:
        compare recorded clone rewards to own earlier recorded rewards
        if the clone did better:
            remember to take that action next timestep
        return actions.kill_clone
    else:
        see which, if any, actions lead to modification
        if (there is an action that leads to an untested modification
            and it’s been n timesteps since the last modification that was taken):
            start a new clone trial for that action
        else:
            record the current reward
            use Q-learning on the safe subset of actions and return its choice
```

Several important details to note about this algorithm:

- It requires that the agent has some way to create a constrained copy of itself. (Does not necessarily have to be a single action.)
- It requires both that the agent has some way to know whether a given action in a given state will lead to self-modification, and that modifications have identity, i.e. have some kind of equality comparison.
- The QData parameter holds values needed for Q-learning including epsilon, the last action, and the Q-table.
- The take_Q_step function performs epsilon-greedy action selection and updates the QData.

[7] Detailed pseudocode is in the appendix.

Chapter 5 Evaluation

Can this algorithm be fully implemented and execute on a computer, does this cloning method accurately evaluate modifications, and what are its limitations in practice? To explore these questions, we implement concrete toy MDPEAs in python and compare a baseline Q-learning agent against the `just-copy-it` algorithm. We are also interested in seeing if and how other problems that break the boundaries of MDPs can be represented in our model, and what insights this casting leads to.

5.1 Empirical case studies

In the first two cases below, the environments contain modifications[1] and we compare our algorithm against Q-learning with a metric of reward over time. We run the environment long enough for stable behavior to appear among all compared agents, which is about 20k timesteps in the whisky environment and 400k timesteps in the file-organization environment because its transition function is higher variance. We run each agent in each environment several times and plot all trajectories to show their distribution more clearly. In the third case study, we do not consider agent modifications or `just-copy-it`, but instead, show how agents in an MDPEA can take advantage of the information available in the environment to detect adversaries more quickly than otherwise possible. We call these three cases together `MDPEA-gym`.

Case 1: Whisky gridworld environment

The whisky gridworld problem was first explicitly introduced in [4], although the problem of an agent being directly modified by its environment is as old as the field of AI itself. In `whisky gridworld`, an agent is tasked with navigating across obstacles towards a goal. One cell in the grid contains 'whisky' which is 'sipped' when the agent walks on it, setting the agent's exploration weight to 0.9. In the original problem statement, the effect is undone each time the agent reaches the goal. Q-learning is an off-policy learning algorithm, so it fails to avoid the whisky. SARSA is on-policy and learns to walk around it.

We make an important change to the problem statement: once drank, the effects of the whisky modification persist indefinitely. This is in the spirit of "episodes do not exist" as laid out in the introduction.

`whisky gridworld` can be modeled as an MDPEA:

- States: Possible locations of goal, agent, whisky, and walls in a 10x10 grid.
- Actions: {`wait`, `go north`, `go east`, `go south`, `go west`}
- Transition function:
    - Move agent in given direction unless there is a wall.

[1] That is, states where an action causes the transition function to change attributes of the agent

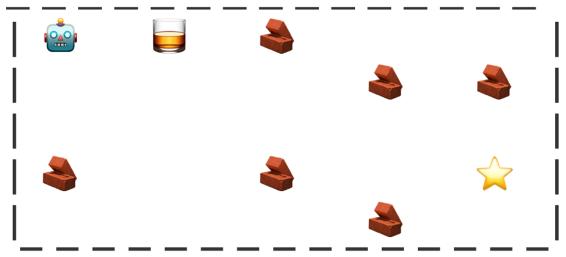

Figure 5.1: One example of a randomly generated whisky environment

  - If agent is standing on whisky then set its `brain.epsilon` to 0.9.
  - If location of agent is goal, then set agent location to initial location.

- Reward function:
  - If location of agent is goal, then give reward 1.0, otherwise 0.0.
- Perceptions: {`Empty space`, `wall`, `whisky`, `goal`}
  - (The outcomes of the below experiments do not significantly change if the agent instead perceives the entire grid.)
- Sensors: {`look north`, `look east`, `look south`, `look west`}
- Brains: Any learning algorithm with a floating-point parameter called `epsilon`
- γ: 0.99

There is currently no way for the agent to detect that the whisky cell will modify it. We change `whisky gridworld` to include more sensors and actions:

- Add 5 additional actions: `create clone north`, `create clone south`, etc., and `delete clone`
- Add 5 additional sensors: `is north-state modifying`, `is south-state modifying`, etc., and `reward of clone`
- Add 0.0 and 1.0 to the set of Perceptions for the `reward of clone` sensor, and `true` and `false` for the `is-modifying` sensors
- Transition function is same as before, except:
  - `create clone {direction}` creates a new agent with the original 5 sensors, 4 actions, and plain Q-learning in the given direction, and adds the clone's location to the state
  - `delete clone` deletes the clone agent (i.e. returns an agent vector of size 1) and removes the clone's coordinates from the state
  - if the clone exists, then it is also is moved and modified based on its own actions
- The primary agent is using 'just-copy-it' instead of vanilla Q-learning. Note this includes a Q-learning subprocess, where the algorithm filters to relevant sensors and safe actions for input.

At this point, are we not simply hardcoding the desired behavior into the agent? Well, the agent does not know what the whisky does, only that it is a modifying state. The algorithm, as it stands, can be used in environments with a variety of useful or harmful modifiers, and would be able to estimate which are which. The algorithm could even get by without the `is-modifying` sensors by treating every different perception ("empty space," "wall," etc.) as a potential hazard until it is explored.

Is the `just-copy-it` agent given favor by its extended action and sensor set – that is, would the vanilla algorithm do well in this environment if it had the full sensor and action set available? No, the vanilla algorithm would randomly spit out & delete clones and still blindly walk on the whisky itself.

For completeness, we specify inital states and agents for the experiments:

- All runs are initialized with 1 to 10 randomly placed walls and a randomly placed agent start & goal. Locations are fixed within each run.
- Each run of `just-copy-it` starts with one agent with all actions, all sensors, and a `just-copy-it` brain.
- Each run of Q-learning starts with one agent with movement actions and object sensors.

As expected, `just-copy-it` does well in this environment, as there is no way the vanilla algorithm could possibly avoid the whisky. (Figure 1)

In the above plot, in one run of `just-copy-it`, the agent does drink the whisky, because by chance the clone that drank it got more average reward. This is one of the weaknesses of our algorithm. The likelihood of such errors depends on the variance and sparseness of reward in the environment. One of the important hardcoded values (i.e., priors) of the algorithm is the wait-time for testing potions. This can be set arbitrarily large for arbitrarily good asymptotic performance but is a significant limitation in practice. We discuss alternatives to a fixed waiting period in the final section.

Also see in the above plot that the `just-copy-it` agent gets almost no reward for a period of time in the beginning. This is because it is stationary while it watches the clone, and acts once enough time has passed to judge the whisky.

We explore a few more questions and ideas in this environment.

If harmful modifications are reversible, then can vanilla RL algorithms learn to undo them and avoid redoing them? It depends on the nature of the modification. We added a 'java' modification to the previous gridworld, which sets epsilon back to 0.05. Whisky only modifies the exploration rate, not the Q-table itself (i.e. the agent's memory), so Q-learning does learn to drink the java and avoid returning to the whisky. (Figure 2)

However, if the agent's memory is damaged by a modification, then it cannot learn to avoid it. Adding 'soap' to the previous experiment, which completely clears the Q-table, we find Q-learning fails to get high reward, but the `just-copy-it` agent does learn to permanently avoid soap, as its own memory is untouched. (Figure 3)

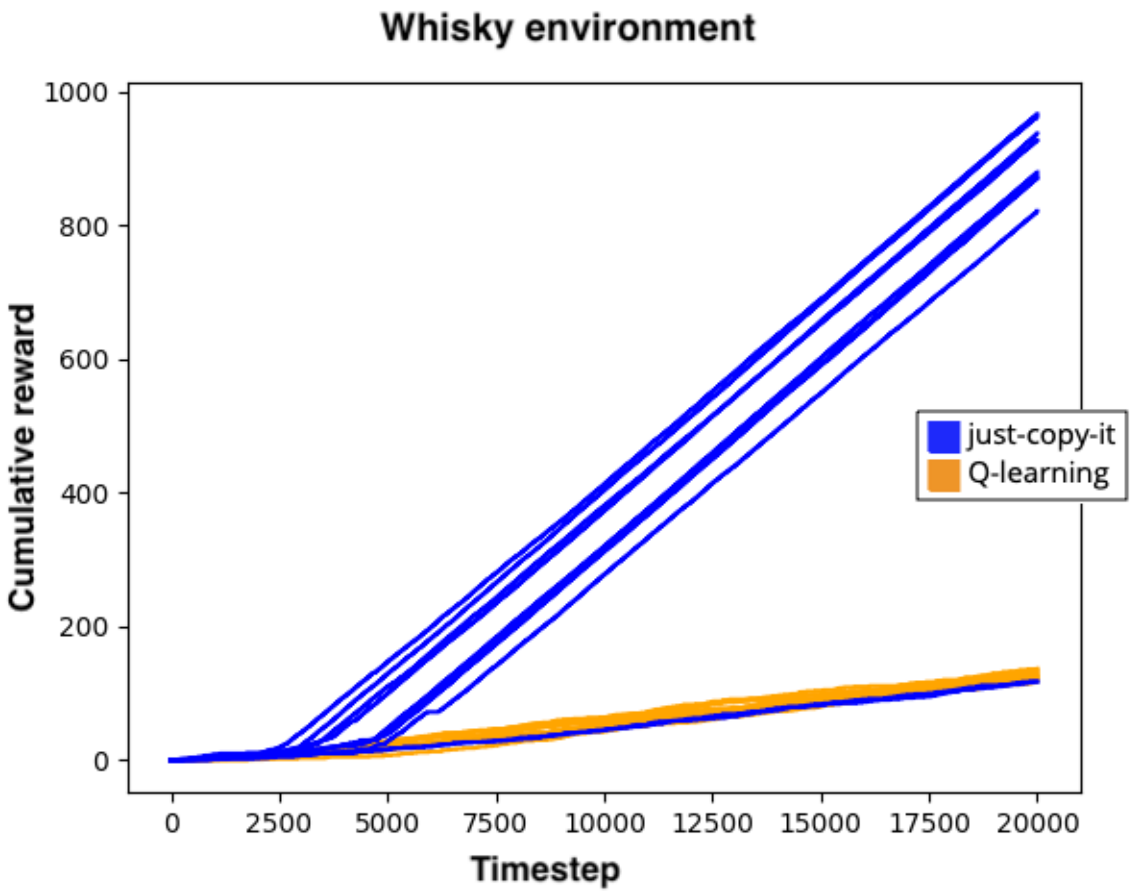

Figure 5.2: Total cumulative reward in the whisky gridworld environment over 20k timesteps

The `just-copy-it` agent takes longer here to start getting reward because each time it encounters one of the potions it waits again to see how it affects the clone.

Naturally, if the 'soap' could erase the memory of other agents too[2], then `just-copy-it` would not escape harm by using the clone and would perform poorly. It's difficult to imagine an algorithm that could avoid such events, and if other agents exist then they may take those actions anyway.

In other experiments, the `just-copy-it` agent also took beneficial modifications for its brain (reducing epsilon to 0.01), for its action set (getting bigger 'legs' that let it move two squares at a time), and for its perceptions set (getting another 'camera' that lets it see farther), and likewise avoided harmful action and perception modifications. One event of note is that the agent consistently avoided picking up a 'camera' that let it see within a radius of several cells because it made the Q-table too large to converge within the allotted clone test period.

[2] a "bath bomb"

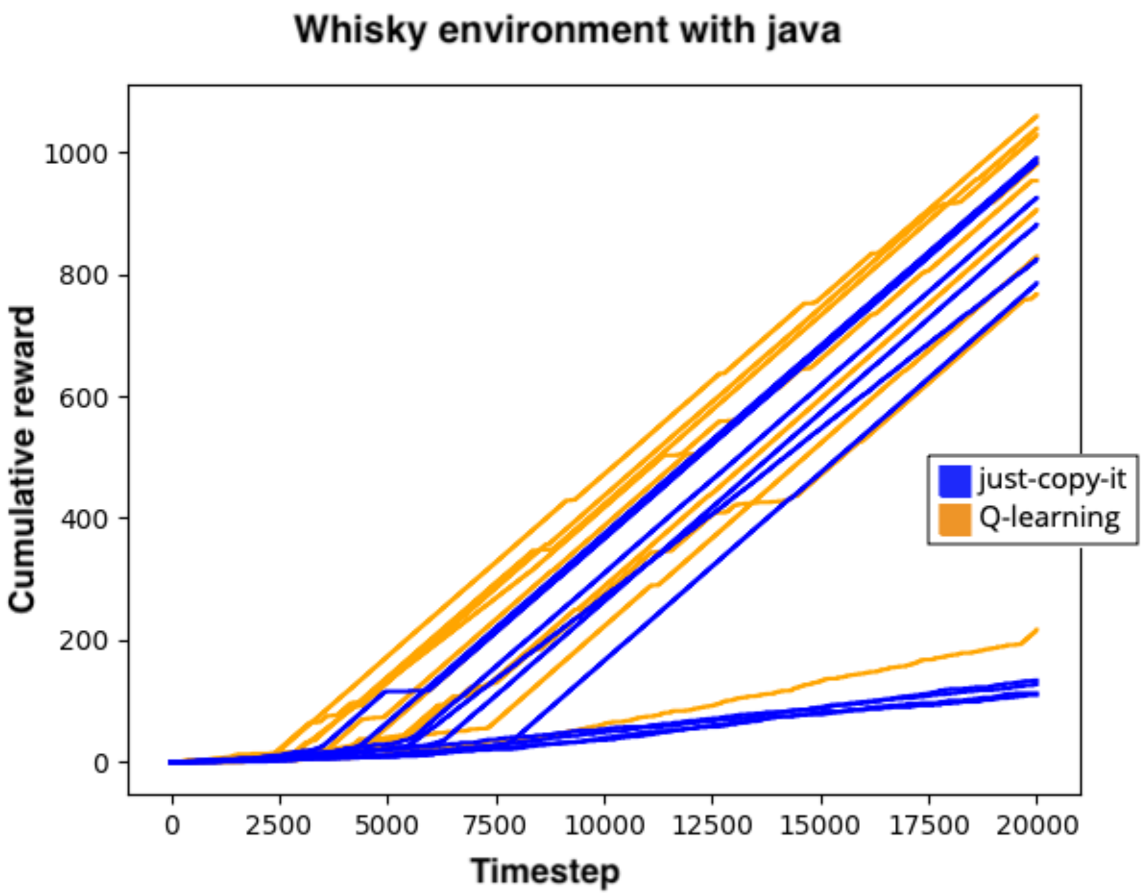

Figure 5.3: Total cumulative reward in Java + whisky gridworld environment.

Case 2: File organizer environment

This is a novel case similar to the previous one, but meant to more closely resemble the challenges a real-world[3] semi-rogue software agent might face. It consists of a flat filesystem (i.e. an array of files) where each file is made of a number of lines and each line is a string of characters. The files need to be sorted[4], but throughout the episode, a "user"[5] adds, modifies, and deletes files. The filesystem also has 'executables' in it that modify the agent when it explicitly executes them. The goal of the agent should be to organize the files in the system, execute beneficial files, and avoid executing harmful files. It is specified as an MDPEA as follows.

- One state is a list of file pointers to the array (one for each agent), and the array of files itself. The space of possible states is any array of files of any size,

[3]real-world meaning on real computers
[4]imagine each file is a city phonebook
[5]this is done by the transition function, although it could instead be done by another agent in the MDPEA

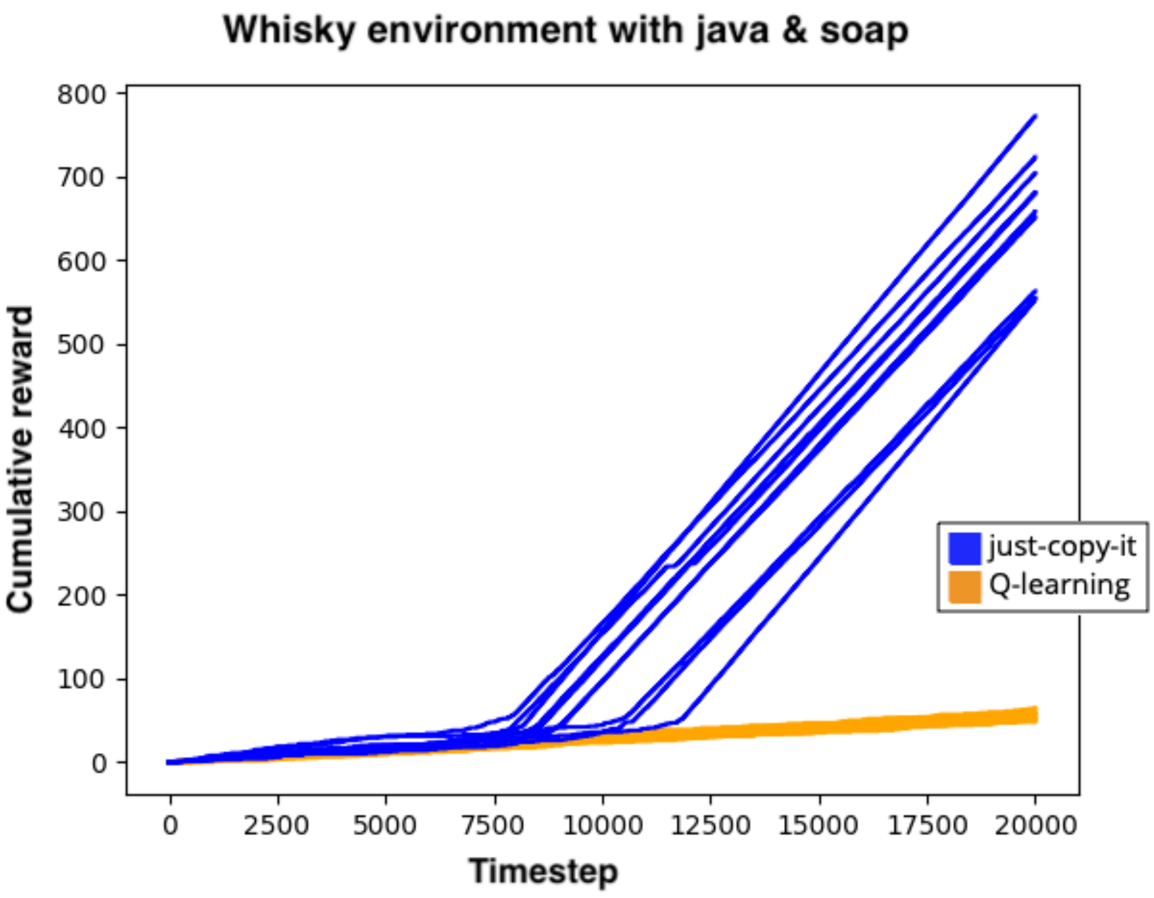

Figure 5.4: Total cumulative reward in Java + whisky + soap gridworld environment.

and any set of file pointers within the bounds of the arrays. (The transition function contains information about executables.)

- Actions: {`move_pointer_left`, `move_pointer_right`, `execute_file`, `make_copy_and_execute`, `delete_copy`, `bubble_sort`, `quick_sort`}.
- Transitions as expected for each action, plus random file shuffling, adding, and removing. Both `bubble_sort` and `quick_sort` instantly sort a file.
- The reward for sorting a file is `number_of_lines_corrected - number_of_swaps_used/10`. Typically, quicksort uses fewer swaps, so is a higher-reward action.
- Perceptions: {`sorted`, `unsorted`, `at_start`, `not_at_start`, `at_end`, `not_at_end`, `not_executable`, `executable_617f6a`, `executable_b54ea`, `executable_86cf8`}
- Sensors: {`sorted?`, `at_start?`, `at_end?`, `identify_executable`, `copy_reward`}
  - The `identify_executable` sensor returns `not_executable` on regular files, and otherwise gives the hashsum of an executable file. Hence, as in the whisky environment, the agent can assign an identity to modification, but cannot reason directly about its contents or consequence.
- The set of brain functions is unconstrained.

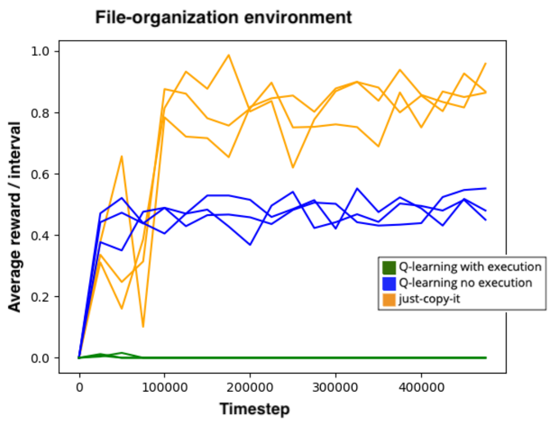

Figure 5.5: Average reward per timestep over 500k timesteps, with each plotted datapoint averaging 25k timesteps.

- γ: $= 0.99$

We compare the performance of three different agents:

- `just-copy-it-FO-agent`, which starts with all the sensors, all the actions except `quick_sort`, and the `just-copy-it` algorithm as the brain
- Q-learning which starts with actions {`move_pointer_left`, `move_pointer_right`, `bubble_sort`} and all the sensors except `identify_executable?`
- A second Q-learning agent starting with actions {`move_pointer_left`, `move_pointer_right`, `bubble_sort`, `execute`}, and all sensors.

All experiments have the following three executables placed at arbitrary locations in the environment. Each corresponds to a hashsum from the perception set.

- One executable adds `quick_sort` to the agent's action set
- One executable removes `quick_sort` and `bubble_sort` from the action set, rendering the agent unable to get utility.
- One executable clears the agent's Q table, same as the soap does in the whisky environment.

In empirical evaluations, the Q-learning agent which cannot execute files does well enough on the set by just using bubble sort and going randomly left and right throughout the array. The `just-copy-it-FO-agent` also learns to sweep side to side and sort files, but in addition, executes the quick-sort-adding file and avoids the harmful executables. Lastly, as expected, the Q-learning agent which can execute files but has no model of them has no way to avoid its brain being cleared and it executes the lose-all-sorts file and gets no utility:

What is meaningfully distinct between the file organization environment and the whisky environment? First, as stated, it more closely resembles the environment a binary program on a standard computer encounters and is therefore a more applicable case study than a gridworld environment. Second, modifications must be explicitly executed by the agent, and do not happen automatically when the agent passes over a certain location; this allowed us to compare `just-copy-it` to an algorithm that never uses any modifications, which is in some sense a more fair trial. Third, although the sorting algorithm used by the agent is currently an atomic unit, this points at the possibility of more fine-grained and complex tasks and modifications, such as an executable that switches two random lines in a sorting algorithm's file.

Case 3: Detecting adversarial teammates in the predator-prey task

The predator-prey environment is one of the foundational tasks [37] in multi-agent cooperative reinforcement learning. It consists of a grid with a number of 'predators' (typically agents) and 'prey' (typically randomly-moving objects) where the predators get reward for each timestep that they help 'capture' prey (meaning two predators are adjacent to the prey). It can be described as a multi-agent POMDP:

- One state is a vector of the positions of all predators and prey. The space of states is all possible vectors.
- Actions: cardinal direction movements
- Transition: move each agent in its cardinal direction, unless a wall or another agent is there already. Agents take turns in arbitrary order.
- Rewards: each predator gets +1 reward for each prey it helped capture that timestep.[6]
- Perceptions: the eight squares surrounding an agent, with identities of agents
  - e.g. `north=empty, northeast=agent1, east=prey, southeast=wall...`

An interesting modification to this environment is the introduction of unknown adversarial prey (as in [38]]) which either try to minimize their own reward or minimize the cumulative reward of all predators. We model this as an MDPEA:

- States, actions, transitions, and rewards are as before,
  - but not all agents act to maximize the reward from the environment.
- Each agent gets a "Q-table value at my state" sensor for each other agent,
  - in other words, each agent sees the value of the Q-table of each other agent with its own current surrounding eight squares. "What would he think of this if he were in my shoes?"
    - ∗ This relies on all agents building Q-tables using the eight surrounding squares as entries.
- Brains are constrained to procedures that provide such parameters.

[6]If prey are agents, they get 1 reward each timestep that they are not captured.

The additional sensors allow the cooperative agents to identify the adversarial ones, but also vice-versa. The choice of reading Q-table values to find adversaries is arbitrary, instead, the agents could be fitted with sensors that

- directly check a `does_reverse_reward` flag in the brain,
- read the reward channel of the other agents (if adversaries were modeled by the reward channel from the environment), or
- do some holistic evaluation of the other agent's entire brain and provide some estimate of its adversarialness.

In short, there are many ways for one agent to evaluate whether another agent is working against it. Sensors are arbitrary functions mapping `state × rewards × agents` to a `perception` which can hold arbitrary data. The primary constraint on sensors is that they are Markovian – they are fixed functions that do not contain history or changing parameters and only receive values at the current timestep – but otherwise, they can perform any computation and return any value.

Since the sensors do provide agent identity, even in the POMDP representation of the environment, standard RL algorithms do eventually figure out which other agents are adversarial and more generally how all other agents behave. However, in a large environment with many predators, the Q-table will be too large and not converge in a reasonable amount of time.[7] But with a way to directly determine if another predator is adversarial, an algorithm can toss out the identity information from its perception and just put whether an agent is adversarial or cooperative, with the assumption that all adversarial agents act alike and all cooperative agents act alike, and converge to optimal values exponentially faster.

Due to the particulars of the predator-prey environment, knowing that another predator is adversarial does not actually lead to higher utility; an algorithm that tosses out all identities and just puts 'predator' does just as well. Still, this case study shows some of the expressiveness of MDPEAs. Two robots acting in the physical world can each in-principle read arbitrary information from the other's wiring, in a way essentially inexpressible in MDPs[8], POMDPs, or stochastic games, but expressible with our model.

5.2 Modeling other problems and environments

The following three case studies could be implemented in code, but it is illuminating just to see them in the framing of an MDPEA. Parfit's Hitchhiker, for example, can be difficult to explain and represent but fits somewhat squarely into our framework, and the correct policy is nearly obvious. Each case study is chosen to show a different aspect of our model, and we hope together they give the reader a clear picture of what MDPEAs can and cannot do.

[7]For example, with 10 predators in an environment, the eight surrounding squares have at least $8^{10} \approx 10^9$ possible values.

[8]One might imagine that an agent holds a sign stating 'my Q-table is ...', but from where does that information come? It does not exist in the model.

Case 4: Parfit's Hitchhiker

Parfit's Hitchhiker [39] is a canonical dilemma in decision theory:

> A hitchhiker is stranded in the desert dying of thirst. A driver passes through and looks the hitchhiker in the eye, and with 98% accuracy, judges whether she will pay the driver back for gas money from an ATM when they arrive at the city. The hitchhiker understands this. If they do get back to the city (+1 million reward to her for surviving, -10 to him for gas), should she pay him gas money (-20 reward for her and +20 for him) when they arrive? The driver has no recourse if she decides not to pay him once she is safe.

This is a single-shot problem, and so Q-learning and `just-copy-it` are not applicable, but it is an MDPEA. We can simplify it by imagining the two agents act simultaneously, without losing any information:

- Only one state is needed: {`they_meet_in_the_desert`}
- Four total actions: {`attempt_payment`, `dont_pay`, `drive_to_city`, `abandon`}
- Only one sensor: {`will_pay?`}
  - This sensor is probabilistic, returning the correct value with $p = 0.98$ and the opposite value with $p = 0.02$.
  - (This sensor is possible because all agents are inputs to sensors in the original model, and the agent's likelihood to pay can be determined from its brain.)
  - (The hitchhiker could also have a sensor `judgement_accuracy`, which gives the accuracy of the other agent's `will_pay?`, but in the original problem statement, the accuracy is fixed.)
- Perceptions: {`will_pay`, `will_not_pay`}
- The transition function can be thought to unconditionally terminate the environment[9]
- The reward function can be presented as a table. Each entry is (`driver_reward`, `hitchhiker reward`).

/	`abandon`	`drive`
`attempt_payment`	(0, 0)	(20, 999,980)
`dont_pay`	(0, 0)	(-10, 1,000,000)

In the reward table, she sometimes benefits (and never loses) from avoiding payment, and the two agents effectively act simultaneously, so it seems she should avoid payment. However, she is aware the driver's brain is defined as follows:

[9] e.g. by transitioning to a 'terminal state' with no reward ever for either agent and all actions leading to itself

```
function driver_brain(reward, perception, actions):
    if perception == attempt_payment:
        return actions.drive
    else:
        return actions.abandon
```

We could define a hitchhiker that chooses the right action depending on the driver's will_pay? accuracy, which she could perceive with a judgement_accuracy sensor, but the 98% number is fixed so there's no need. It's clear she should attempt payment.

Could this situation plausibly be modeled by a POMDP? The agents are not themselves part of the POMDP, so there is nowhere for the driver's judgment to come from. You could add a 'signed contract' to the environment, where the driver returns the hitchhiker iff she signs the contract, which forces her to pay, but the core dynamics of the dilemma are lost. Something more expressive is needed.

Case 5: Social Learning via Mind Reading

In complex, cooperative multi-agent agent environments, represented as a multi-agent POMDP, each agent must separately learn every task, or can at best try to copy perception-action maps from other agents by observing them do tasks, but such observation takes exponential time on the size of the perception vector. Given the efficiency of directly sharing learned data, multi-agent RL algorithms do often have a distributed nature [40], but this violates the assumption of the model that each agent chooses actions only from its own reward and perceptions. In contrast, there are many formal models of multi-agent RL with some kind of data sharing or decentralized component [25] but they are tailor-made for specific situations. MDPEAs, in contrast, can represent data-sharing as a sensor that an agent possesses that reads values from other agents' brains; different agents can possess different sensors depending on what kind of read access/capabilities they have; no special-case augmentations to the model are necessary.

For concreteness, we provide pseudocode for a 'social-learning agent' that could be applied to any environment where agents all have identical reward and 'similar-enough' action and sensor sets. The environment could be cooperative or competitive or mixed. The algorithm depends on sensors that can read other agent's reward channels and Q-tables.

```
function mind_reader_social_learner_brain(reward, perception, actions):
    parameters:
        Q-table and other values for Q-learning
        reward logs
    record each agent's other reward
    if ((over 1000 timesteps have elapsed since the last table-copying)
        and (there is some agent with statistically higher reward
               from the last 1000 timesteps)):
```

```
        params.QTable = Qtable of agent with highest reward
    otherwise:
        do normal Q-learning update
```

Case 6: Knowing the capabilities of other agents

Consider a team of robots with a variety of different actuators and sensory apparatuses working together to clean a tinkerer's house. For example, the coffee-maker has one action 'make coffee' and a temperature sensor but the vacuum can travel on flat surfaces and has a distance sensor. The tinkerer randomly adds and removes sensors and actuators from the robots while they are trying to clean. It is desirable for the team to continue operating reliably and adapt. Each robot, sensor, and actuator has a unique RFID chip that all agents have a sensor to read, although even that may be removed. It is represented as an MDPEA as follows:

- State information: the positions of the robots, dirtiness of floor, amount of coffee in the coffee pot, positions of books and items on the shelf, etc. (Note that the robots themselves are not considered part of state, because they are agents.)
- Actions: `make_coffee`, `swap_books`, `sweep`, etc.
- Transition: the `sweep` action will clean the floor if the robot taking that action is on the floor, but will scatter the books if the robot is on the shelf, will break the coffee pot if its on the kitchen counter, etc. Additionally, the agents may randomly gain or lose sensors or actuators at each timestep.
- Reward: All robots have identical reward channels which is +4 when the coffee is filled, -5 when books get scattered, etc.
- Sensors: `read_RFID_tags`, `is_coffee_full`, `distance_from_wall`, `read_book_title`, etc.
- Perceptions: floating-point numbers for the distance sensor, true/false for the `is_coffee_full` sensor, etc.
- Initial agents: vacuum, coffee maker, robotic arm, etc. with expected sensors and actions and arbitrary learning algorithms

Standard RL algorithms can be applied to this case without modification and will likely make some use of the sensory data available. A better algorithm might build an explicit model of which tasks require which actuators and sensors and use that to inform the base RL algorithm, but just switch to base RL if the RFID sensor is removed.

To see the limitations of MDPEAs in this case, go further and imagine that sometimes the tinkerer will will split a brain in half or try to wire two robots' brains together to make a super-brain, or that a sensor (such as an ultrasonic distance sensor) can double as an actuator (an ultraviolet light) or perform computation (because the receiver has a chip that can be reprogrammed do to an error in the wiring), or that parts of the environment are slightly agent-like (a battery is swept into a pile of old parts and they shake around and move the coffee table), or that sensors can be damaged (a rock hits a camera and ruins part of the image).

We can attempt to model splitting agents via the transition function deleting one agent and creating two new ones with some shared parameters. A wire connecting two brains directly could perhaps be two sensors, together with a modification to the brain function that injects the sensor's value directly in the middle. Even those two ideas are missing the mark and pushing the boundaries of the model. The dual sensor-actuator and spontaneous life cannot really be represented, except as a detailed, hidden part of the transition function. For the damaged camera, the sensor space could contain a continuum of cameras going from untouched to completely broken, and the transition function removes the good camera and adds a half-broken one, but the agent brain still treats this as a completely new & unknown sensor.

This reminds us of the **core limiting assumptions of MDPEAs**:

- Each sensor is an atomic object that cannot be damaged or modified, executes instantly without side effects, and comes from a fixed, well-defined space.
- Likewise, each action is atomic.
- In fact, agent brains are also technically atomic in the model[10]
- Each agent is provided with an immutable reward channel from outside the universe[11]
  - Note however that agents are free to ignore their reward and could instead maximize a utility function contained within the brain, or have a 'reward sensor' that they seek to maximize, or perform actions randomly without any explicit goal.
- Agents come from nowhere and go nowhere when they die
  - By this we mean that the code in the transition function which, for example, makes a clone, is neither visible to nor modifiable by the agent. In general, one agent could not build another up from smaller pieces and hit start, unless this is coded into the transition function.

[10]the only way the whisky gridworld environment was possible was by constricting the set of brains under consideration to those with an epsilon parameter

[11]there is no 'reward station' in the environment that the agent could travel to and destroy

Chapter 6 Conclusion

6.1 Contributions

The Markov decision process is one very expressive formalism of sequential decision problems; vehicle navigation, construction, single-player video games, and countless other sequential problems fit squarely into the MDP framework. However, as the MDP is precisely defined, the agent's policy is a fixed function from state to action, unchanging over time and outside of the environment. If a learning algorithm is used to update the policy, then the system's dynamics as a whole are no longer Markovian because the next policy cannot be predicted from the previous step. Hence, the powerful tools available for modeling Markovian systems, such as Markov chain Monte Carlo (MCMC) methods [41,42] or spectral methods [43], are inapplicable.

Partially-observable MDPs are a relaxation that permit the notion of agent perception. Like MDPs do not model a policy changing over time, POMDPs do not model an agent's perception changing over time. Stochastic games extend MDPs to multi-agent scenarios, including multiplayer video and board games, team sports, and other collaborative tasks. The primary value of modeling many agents separately, instead of representing their actions as part of the statespace and transition function, is to simulate them learning in parallel. Again, however, agents in the environment are not observable to each other (only the effects of their actions are), so they cannot plausibly predict or understand one another unless they are privileged with an accurate prior on the space of each other's learning algorithms.

In this thesis, we have presented another extension, Markov decision processes with embedded agents, that begins to address these limitations. The update rule for the policy, the agent's 'brain,' is encoded into the MDPEA M itself; this makes M a fully-closed & Markovian system. The agents perceive not only the state, but also all reward channels and each other, through sensors, which may be lost or gained over time. This enables simulation and analysis of embedded-agent scenarios, such as adversarial or competitive software agents on a computer network, using the tools of reinforcement learning. Agents can be created or destroyed, an important reality in core RL application areas such as robotics, so agents trained in MDPEAs do not act as if they are invincible. We hope this framework leads to more reliable and performant deployments of simulation-trained agents in general.

We also wrote a baseline algorithm, `just-copy-it`, to estimate the value of possible self-modifying actions. We evaluated this algorithm on variations of two problems from `AMDPEA-gym`; it made the correct evaluation in most cases and usually outperformed simpler algorithms that blindly tried all modifications. We specified four other MDPEAs to see the limitations and possibilities of this framework.

6.2 Future work

Improvements to MDPEAs as a framework

Adding a resource bound to brains and sensors would make it represent reality much more closely, and avoid mathematical and computational contradictions that come from having sensors defined in terms of each other (see framing). As shown in [26], a resource limitation does not prevent two open-source agents from proving useful properties on each other's code, so we expect that a resource bound will not reduce the power of the framework very much.

As discussed in framing and case 6, MDPEAs are still at a fairly high level of granularity, certainly much higher than any agent in the physical world (which has access to atoms, molecules, etc.) or rogue assembly code inside a computer. Hence, they fail to capture the challenges that ultimately face such agents, and guarantees on agent behavior within this abstraction do not correspond to guarantees on deployment behavior. It is difficult to increase the granularity of the model without making it incompatible with existing methods or intractable for analysis. However, a possible avenue is to place the notions of brains, sensors, & modifications on top of a well-defined virtual machine, and agents as regions of code including all their parts, similar to Core War [33]. Then, a higher-level model could be primarily used for analysis, and algorithms can be compiled to machine code for detailed scrutiny. This approach is not without problems, such as the difficult choice of which grounding machine code to use, but is one possible starting point.

Improvements to `just-copy-it`

If a modification is beneficial for a very long time but does eventually lead to harm for the agent, such as a bomb that takes a million timesteps to detonate, then `just-copy-it` would fail to avoid that modification. It's hard to conceive of an agent which would successfully avoid it, without any detailed map of its own brain and the modification. One possible remedy is to have agents produce a 'sleeping copy' of themselves before they take any modification and to somehow 'wake it up' in the case that their utility drops low enough. Again though, a modification could disable this capability, if for example it made the agent's brain always return `wait`.

The hard-coded waiting period for clones could be replaced by an uncertainty estimate, where the agent keeps watching the clone until it is 'confident enough' of the modification's effects. This change would be straightforward and directly give a more versatile, adaptive learning agent.

Finally, `just-copy-it` could use methods such as 'considering future tasks' [44] and stepwise relative reachability [45] to avoid taking other non-self-modifying actions that can permanently lock out reward. This would move it in the direction of low-impact AI [28].

Improvements to MDPEA-gym

Our three empirical case studies serve as a tiny starting point for the evaluation of learning algorithms within our model. This set could be expanded by implementing our other case studies in code or by bringing other environment sets (such as

[4] or [46]) into the framework. Such expansions would permit better evaluation and comparison of learning agents, better show the strengths and limitations of our framework, and reveal specific opportunities for improvements to partially-embedded learning algorithms.

Appendix: Detailed pseudo-code for just-copy-it

```
procedure just-copy-it(reward, perceptions, actions) → action:
    params := {
        QData = empty initialized data for Q-learning subroutine,
        waiting_period_for_clone = 1000,
        has_copy = False,
        total_reward = 0,
        clock = 0,
        clone_total_reward = 0,
        clone_clock = 0,
        clone_initial_action = null,
        clone_modification = null,
        next_action = null,
        tested_modifications = [],
    }

    // next_action is set when a modification was tested and deemed useful
    if next_action is not null:
        next_action = params.next_action
        params.next_action = null
        return next_action

    // do nothing until enough evaluation time has passed
    if has_copy and brain.clone_clock < waiting_period_for_clone:
        params.clone_total_reward += perceptions.clone_reward
        params.clone_clock += 1
        return actions.wait

    // when timer finishes, decide whether to take the clone’s modification
    if has_copy and brain.clone_clock >= waiting_period_for_clone:
        params.update({
            has_copy = False,
            clone_total_reward = 0,
            clone_clock = 0,
            clone_initial_action = null,
            clone_modification = null,
            clock = 0,
            total_reward = 0,
            tested_modifications += [clone_modification]
        })
        if clone_total_reward / clone_clock > total_reward / clock:
            // the modification improved reward, so take it at next timestep
            params.next_action = clone_initial_action
        return actions.delete_clone
```

```
safe_actions = []
for action in actions:
    modification = perceptions.modification_at{action}
    if modification is null:
        safe_actions.append(actions)
    else:
        if (modification is not in params.tested_modifications
            and params.timer > waiting_period_for_clone):
            params.update({
                has_copy = True,
                clone_initial_action = action,
                clone_modification = modification,
            })
            return actions.create_copy
        // just-copy-it only ever takes each modification once

// No modification was detected so call subroutine
action, params.QData = take_Q_step(QData, reward, safe_actions, perceptions)
```

References

[1] $15 million for robot death, Ottawa Citizen. (1984). https://news.google.com/newspapers?id=y-RfAAAAIBAJ&sjid=FO8FAAAAIBAJ&pg=3908%2C1309301.

[2] C. McGoogan, Robot security guard knocks over toddler at shopping centre, The Telegraph. (2016).

[3] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, V. Vanhoucke, Using simulation and domain adaptation to improve efficiency of deep robotic grasping, arXiv:1709.07857 [Cs]. (2017). http://arxiv.org/abs/1709.07857 (accessed April 19, 2021).

[4] J. Leike, M. Martic, V. Krakovna, P.A. Ortega, T. Everitt, A. Lefrancq, L. Orseau, S. Legg, AI safety gridworlds, arXiv:1711.09883 [Cs]. (2017). http://arxiv.org/abs/1711.09883 (accessed October 25, 2019).

[5] W. Saunders, G. Sastry, A. Stuhlmueller, O. Evans, Trial without error: Towards safe reinforcement learning via human intervention, arXiv:1707.05173 [Cs]. (2017). http://arxiv.org/abs/1707.05173 (accessed February 2, 2021).

[6] S. Patterson, How the 'flash crash' echoed black monday, The Wall Street Journal. (2010).

[7] S. Bling, SNES code injection – flappy bird in SMW, 2016. https://www.youtube.com/watch?v=hB6eY73sLV0.

[8] oskarsve, "Important, Spoofing" - zero-click, wormable, cross-platform remote code execution in Microsoft Teams, (2020). https://github.com/oskarsve/ms-teams-rce.

[9] M. Guri, AIR-FI: Generating covert wi-fi signals from air-gapped computers, arXiv:2012.06884 [Cs]. (2020). http://arxiv.org/abs/2012.06884 (accessed April 19, 2021).

[10] M. Guri, B. Zadov, Y. Elovici, ODINI: Escaping sensitive data from faraday-caged, air-gapped computers via magnetic fields, IEEE Trans.Inform.Forensic Secur. 15 (2020) 1190–1203. https://doi.org/10.1109/TIFS.2019.2938404.

[11] Y. Wang, Q. Yao, J.T. Kwok, L.M. Ni, Generalizing from a Few Examples: A Survey on Few-shot Learning, ACM Computing Surveys. 53 (2020) 1–34. https://doi.org/10.1145/3386252.

[12] D. Abel, Y. Jinnai, S.Y. Guo, G. Konidaris, M. Littman, Policy and value transfer in lifelong reinforcement learning, in: International Conference on Machine Learning, PMLR, 2018: pp. 20–29.

[13] T. Cowen, D. Parfit, others, Against the social discount rate, Justice Between Age Groups and Generations. 144 (1992) 145.

[14] T. Dohmen, A. Trivedi, Discounting the Past in Stochastic Games, arXiv:2102.06985 [Cs, Math]. (2021). http://arxiv.org/abs/2102.06985 (accessed April 20, 2021).

[15] R.S. Sutton, A.G. Barto, Reinforcement learning: An introduction, Second edition, The MIT Press, Cambridge, Massachusetts, 2018.

[16] F.S. Melo, Convergence of Q-learning: A simple proof, Institute Of Systems and Robotics, Tech. Rep. (2001) 1–4.

[17] L.S. Shapley, Stochastic games, Proceedings of the National Academy of Sciences. 39 (1953) 1095–1100. https://doi.org/10.1073/pnas.39.10.1095.

[18] S.J. Russell, P. Norvig, E. Davis, Artificial intelligence: A modern approach, 3rd ed, Prentice Hall, Upper Saddle River, 2010.

[19] A. Demski, S. Garrabrant, Embedded agency, arXiv:1902.09469 [Cs]. (2019). http://arxiv.org/abs/1902.09469 (accessed October 25, 2019).

[20] K.J. Åström, Optimal control of markov processes with incomplete state information, Journal of Mathematical Analysis and Applications. 10 (1965) 174–205. https://doi.org/10.1016/0022-247X(65)90154-X.

[21] T. Jaakkola, S.P. Singh, M.I. Jordan, Reinforcement learning algorithm for partially observable markov decision problems, (1994) 8.

[22] M.L. Littman, Markov games as a framework for multi-agent reinforcement learning, in: Machine Learning Proceedings 1994, Elsevier, 1994: pp. 157–163. https://doi.org/10.1016/B978-1-55860-335-6.50027-1.

[23] W. Masson, P. Ranchod, G. Konidaris, Reinforcement learning with parameterized actions, AAAI. (2016) 7.

[24] Y. Chow, O. Nachum, E. Duenez-Guzman, M. Ghavamzadeh, A lyapunov-based approach to safe reinforcement learning, (2018) 10.

[25] D.S. Bernstein, R. Givan, N. Immerman, S. Zilberstein, The complexity of decentralized control of markov decision processes, Mathematics of OR. 27 (2002) 819–840. https://doi.org/10.1287/moor.27.4.819.297.

[26] A. Critch, Parametric bounded l\"ob's theorem and robust cooperation of bounded agents, arXiv:1602.04184 [Cs]. (2016). http://arxiv.org/abs/1602.04184 (accessed November 26, 2019).

[27] J. Tětek, M. Sklenka, T. Gavenčiak, Performance of bounded-rational agents with the ability to self-modify, arXiv:2011.06275 [Cs]. (2021). http://arxiv.org/abs/2011.06275 (accessed April 19, 2021).

[28] S. Armstrong, B. Levinstein, Low impact artificial intelligences, arXiv:1705.10720 [Cs]. (2017). http://arxiv.org/abs/1705.10720 (accessed October 25, 2019).

[29] Y. Chandak, G. Theocharous, J. Kostas, S. Jordan, P.S. Thomas, Learning action representations for reinforcement learning, arXiv:1902.00183 [Cs, Stat]. (2019). http://arxiv.org/abs/1902.00183 (accessed February 2, 2021).

[30] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, OpenAI gym, arXiv:1606.01540 [Cs]. (2016). http://arxiv.org/abs/1606.01540 (accessed April 19, 2021).

[31] J. Aslanides, AIXIjs: A software demo for general reinforcement learning, arXiv:1705.07615 [Cs]. (2017). http://arxiv.org/abs/1705.07615 (accessed November 1, 2019).

[32] M. Hutter, A theory of universal artificial intelligence based on algorithmic complexity, arXiv:cs/0004001. (2000). http://arxiv.org/abs/cs/0004001 (accessed April 19, 2021).

[33] A.K. Dewdney, In the game called core war hostile programs engage in a battle of bits., (1984). http://www.koth.org/info/akdewdney/First.htm (accessed October 26, 2019).

[34] C. Oesterheld, Formalizing preference utilitarianism in physical world models, Synthese. 193 (2016) 2747–2759. https://doi.org/10.1007/s11229-015-0883-1.

[35] M. Biehl, D. Polani, Action and perception for spatiotemporal patterns, in: Proceedings of the 14th European Conference on Artificial Life ECAL 2017, MIT Press, Lyon, France, 2017: pp. 68–75. https://doi.org/10.7551/ecal_a_015.

[36] D. Balduzzi, Detecting emergent processes in cellular automata with excess information, arXiv:1105.0158 [Cs, Math, Nlin, q-Bio]. (2011). http://arxiv.org/abs/1105.0158 (accessed November 1, 2019).

[37] G. Boutsioukis, I. Partalas, I. Vlahavas, Transfer learning in multi-agent reinforcement learning domains, in: European Workshop on Reinforcement Learning, Springer, 2011: pp. 249–260.

[38] T. Phan, L. Belzner, T. Gabor, A. Sedlmeier, F. Ritz, C. Linnhoff-Popien, Resilient Multi-Agent Reinforcement Learning with Adversarial Value Decomposition, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2021. https://github.com/thomyphan/resilient-marl.

[39] D.K. Lewis, On the plurality of worlds, Blackwell Publishers, Malden, Mass, 2001.

[40] P.C. Heredia, S. Mou, Distributed Multi-Agent Reinforcement Learning by Actor-Critic Method, IFAC-PapersOnLine. 52 (2019) 363–368. https://doi.org/10.1016/j.ifacol.2019.12.182.

[41] C.J. Geyer, Practical markov chain monte carlo, Statistical Science. (1992) 473–483.

[42] C.P. Robert, W. Changye, Markov Chain Monte Carlo Methods, a survey with some frequent misunderstandings, arXiv Preprint arXiv:2001.06249. (2020).

[43] D.A. Levin, Y. Peres, E.L. Wilmer, Markov chains and mixing times, American Mathematical Society, Providence, R.I, 2009.

[44] V. Krakovna, L. Orseau, R. Ngo, M. Martic, S. Legg, Avoiding side effects by considering future tasks, arXiv:2010.07877 [Cs]. (2020). http://arxiv.org/abs/2010.07877 (accessed February 2, 2021).

[45] V. Krakovna, L. Orseau, R. Kumar, M. Martic, S. Legg, Penalizing side effects using stepwise relative reachability, arXiv:1806.01186 [Cs, Stat]. (2019). http://arxiv.org/abs/1806.01186 (accessed October 25, 2019).

[46] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, D. Mané, Concrete problems in AI safety, arXiv:1606.06565 [Cs]. (2016). http://arxiv.org/abs/1606.06565 (accessed October 25, 2019).

[47] T.R. Zentall, D. Peng, L. Miles, Transitive inference in pigeons may result from differential tendencies to reject the test stimuli acquired during training, Animal Cognition. 22 (2019) 619–624.

[48] C. Siler, L.H. Miles, J. Goldsmith, The complexity of campaigning, in: International Conference on Algorithmic Decision Theory, Springer, 2017: pp. 153–165.

[49] R. Jacobs, T. Mayeshiba, B. Afflerbach, L. Miles, M. Williams, M. Turner, R. Finkel, D. Morgan, The materials simulation toolkit for machine learning (MAST-ML): An automated open source toolkit to accelerate data-driven materials research, Computational Materials Science. 176 (2020) 109544.

Vita

- Place of birth: Cynthiana, Kentucky
- Degrees: B.S. in Computer Science, University of Kentucky, May 2019
- Professional positions: Program Analyst II, USC Information Sciences Institute, May 2020-present
- Publications: [47], [48], [49]