WATTS: INFRASTRUCTURE FOR OPEN-ENDED LEARNING

Aaron Dharna
New Jersey Institute of Technology
aadharna@gmail.com

Charlie Summers
Columbia University
cgs2161@columbia.edu

Rohin Dasari
New York University
rd2893@nyu.edu

Julian Togelius
New York University
julian@togelius.com

Amy K. Hoover
New Jersey Institute of Technology
ahoover@njit.edu

ABSTRACT

This paper proposes a framework called Watts for implementing, comparing, and recombining open-ended learning (OEL) algorithms. Motivated by modularity and algorithmic flexibility, Watts atomizes the components of OEL systems to promote the study of and direct comparisons between approaches. Examining implementations of three OEL algorithms, the paper introduces the modules of the framework. The hope is for Watts to enable benchmarking and to explore new types of OEL algorithms. The repo is available at https://github.com/aadharna/watts

1 INTRODUCTION

One approach to designing systems that continue to produce interesting or novel forms (i.e., open-ended evolution or learning; (OEL)) (STANDISH, 2003; Team et al., 2021; Lehman & Stanley, 2011; Schmidhuber, 2013) is co-evolving agents and the problems that they are trying to solve. The paired open-ended trailblazer algorithm (POET) Wang et al. (2019; 2020); Dharna et al. (2020) co-evolves bipedal agents and their environments, represented as terrains that the agents must learn to navigate. Through open-ended evolution, environments are proposed that POET-agents can solve more efficiently than agents trained through standard optimization techniques like Proximal Policy Optimization (PPO) (Schulman et al., 2017). In PAIRED (Dennis et al., 2020) game-playing and game-building agents co-optimize together creating a curriculum of 2D mazes resulting in agents that generalize better than agents trained with Minimax or Domain Randomization techniques for curriculum generation. Team et al. (2021) recently proposed some methods where a single artificial agent can solve 37 games it has never seen before with behavior that generalizes even to held-out tasks.

While measures aim to quantify the open-endedness of such systems (Wang et al., 2020), at a minimum directly comparing the algorithms first requires testing them in the same domain. Bipedal agents in POET navigate 2-D terrains in the Hardcore BipedalWalker-v3 environment, where stumps, gaps, and stairs create obstacles for the agent to maneuver around (Brockman et al., 2016). PAIRED (Dennis et al., 2020) trains agents to solve 2-D mazes from the MiniGrid suite (Chevalier-Boisvert et al., 2018) and DeepMind trains agents to solve game rules and environment topologies in their proprietary XLand Universe Team et al. (2021). Decoupling these algorithms from their environments is often challenging. Beyond that, research code is hard to read which can lead to algorithmic misunderstandings (Dharna et al., 2020; Dennis et al., 2020).

Some little accidents are indeed happy (Ross, 2017); PAIRED (Dennis et al., 2020) proposes a novel, adversarial framework for simultaneously optimizing game-playing and game-building agents. Rather than implementing a filtering mechanism for candidate levels based on a minimal threshold of quality (i.e., the minimal criterion), PAIRED re-conceptualizes the notion as that of the Min player in a game of Minimax Regret (Dennis et al., 2020; Jiang et al., 2021b;a; Parker-Holder et al., 2022) but also incorrectly casts POET as a minimax algorithm. In Team et al. (2021) candidate environments are filtered based on satisfying a minimal threshold of quality similar to the minimal

criterion described in POET. Like PAIRED, the feedback from multiple agents are considered when filtering candidate levels. However, code from (Team et al., 2021) is not yet publicly available, and could prove susceptible to reinterpretations if it were.

This paper proposes a framework called Watts for designing reproducible and comparable experiments in open-ended learning. Motivated by modularity and algorithmic flexibility, Watts atomizes the components of OEL systems to promote the study of and the direct comparisons between OEL approaches. Grid-world learning environments can be developed and customized with the game-engine, Griddly (Bamford et al., 2020), designed with single and multiple agents in mind. Experiments can also be run on any of the learning environments already available in Open AI Gym (Brockman et al., 2016). A distributed library for agent optimization called RLlib is used for single or multi-agent learning (Liang et al., 2018), which facilitates the design of custom algorithms or use of any of those currently available at `https://docs.ray.io/en/latest/rllib-algorithms.html`(Liang et al., 2018). While the innovation sought after in all of these open-ended learning paradigms can arise from cooperative or competitive interactions between agents and/or their environments, the challenge of open-ended learning lies in designing systems that can push search away from the boundaries of convergence toward areas of divergent intrinsic dynamics. To promote this research direction, we have open-sourced Watts and encourage the community to contribute.

To summarize our contributions:

- We propose a modular framework - Watts - to build OEL algorithms with re-usable sub-components.
- We implement POET and PAIRED in Watts and compare them with one another.

Section 2 details design considerations. Section 3 describes Watts' current modules. Section 4 describes how section 3's components can combine into OEL algorithms and how interpreting these pieces in different lights can result in related new algorithms. Section 5 compares these algorithms in a common suite of environments and section 6 discusses initial performance. Section 7 is results and future directions for Watts.

2 Design Considerations and Motivations

At the heart of popular frameworks like Open AI Gym is the notion that standardization enables good science (Brockman et al., 2016). In software engineering and in particular in reinforcement learning, encapsulation can promote code reuse and encourage readability such that errors in these complicated, stochastic, and distributed algorithms are quickly noticed and corrected (Liang et al., 2018). One source of programmatic complexity in designing OEL systems is the large number of distinct modules with interlocking parts. When (co)evolutionary systems manage large populations of learners, they often have components for optimizing agents (Salimans et al., 2017; Schulman et al., 2017), generating environments (Khalifa et al., 2020), evolving tasks dynamically (Team et al., 2021), filtering which candidate individuals can enter the population (i.e., minimal criterion (Brant & Stanley, 2017)), and transferring knowledge from a source learner to a target learner (Wang et al., 2019). Each of these components represent modules that are candidates for code reuse.

Good encapsulation can also promote good code structure, enabling a model of these algorithms as arrangements of their basic building blocks. Designing algorithms is an inherently creative process (Stanley, 2018), and distilling these algorithms into their component blocks can enable combinatorial creativity (Boden, 2009; Paszke et al., 2019). Accessing multiple OEL approaches in a common framework helps illuminate the combinations of algorithmic elements yet to be imagined. Watts provides an abstraction that allows developers and researchers to focus on defining new modules without worrying about the complexity of the rest of the OEL algorithm.

3 The Watts Framework

Watts is designed to conceptualize algorithms as modular pieces that can be mixed, matched, and recombined in novel ways. Shown in Figure 1 are modularized components of OEL algorithms. Components for generating environments are called *Generators*. Components that implement filtering are called *Validators*. *Solvers* define how agents are optimized in Watts. Components that

interact with other elements of the outer loop are implemented as *Strategies*, like an *Evolutionary Strategy* or *Transfer Strategy*. Such modularity enables a "separation of responsibilities" in implementation details to isolate hypotheses.

3.1 Environment Generation

Environments in Watts are created with *Generators*. These generators are modules that enable any number of environment representations (e.g., indirect encodings, direct encodings, wave function collapse). Encapsulating environment generation to its own module enables a common API between new but algorithmically unrelated approaches to procedural content generation in OEL systems.

A generator that creates 2D grid-worlds could store direct (x, y) locations of the objects in the environment (e.g. player locations, wall tiles, traps, and treasure locations, etc.) and the rules that govern increase or decreasing amounts of these objects and/or rearrange their locations (e.g. remove a treasure; move the exit) – this is how POET and PINSKY work (Wang et al., 2019; Dharna et al., 2020). When a generator is queried, a level is produced and variety is obtained by maintaining a population of distinct generators.

Alternatively, generators could store neural networks that sequentially place objects on a blank canvas, such that a new level is created every time the generator is queried (e.g., a new maze) – this is how PAIRED and PCGRL handle level generation (Dennis et al., 2020; Khalifa et al., 2020). And generators can operate on anything between these two extremes. A hypothetical generator from (Team et al., 2021) could create a level topology with Wave Function Collapse (Gumin, 2016) and then query compositional pattern producing networks to place the remaining game-objects. In general, the environment generators can be thought of as parameterizing the missing information of an Underspecified Partially Observable Markov Decision Process (UPOMDP) (Dennis et al., 2020).

3.2 Evolution and Minimal Criteria

An open-ended learning process should be able to shift the training distributions and objectives, and the field of evolutionary computation provides a natural approach to incrementally shifting distributions. The traditional view of an evolution strategy (ES) is one of being a zeroth order optimization algorithm for real-valued parameter-vectors. In an ES rather than calculate (stochastic) gradients of a loss function with respect to the model parameters, gradients are estimated from stochastic perturbations (sampled from an underlying noise distribution, usually a Normal) around the current parameter value. The key insight of this view is that the noise distribution itself can also be learned using feedback of the reward/fitness function e.g. CMA-ES (Hansen, 2007). Because an ES can contain a tunable representation of the underlying probability distribution, ES's are also generative algorithms.

An ES functioning as a generative algorithm provides a method of shifting agent training distributions (e.g. levels in a game). However, unlike using an ES to solve a black-box optimization problem, open-ended systems might require that the training distribution drift through parameter space without a target to optimize towards. Such schemes are called minimal criteria (MC) and represent an alternative to conventional fitness-based paradigms (Soros, 2018) since the MC encodes a task-agnostic measure e.g. how similar is an agent to its neighbor (Soros et al., 2016). In Watts, MCs are represented as *Validators*. Under this premise, so long as a minimum threshold on the quality of samples drawn from the training distribution is met, a candidate solution is accepted into the population of potential answers; the distribution does not change to explicitly optimize with respect to an evaluation metric.

MCs are of particular interest because they encode an undirected method of slow learning (i.e. evolution) to complement the fast learning of agent optimization (Botvinick et al., 2019) resulting in easy to implement world-agent coevolutionary structures. To act as an outer-loop and shift populations, Watts provides an *evolutionary* API. The evolutionary API accepts and returns a population, executing logic that selects parents (i.e., select k random parents; select the k highest performers, etc.), mutates members of the population, and kills population members (i.e., kill the k oldest members of the population if population size is exceeded). The evolutionary API can explore both traditional- and MC-based evolutionary structures as outer-loops on agent-learning.

3.3 AGENT LEARNING

In Watts the class that decouples the specific OEL implementations from the optimization algorithm is the *Solver*. A Solver specifies the algorithm to optimize agent performance. The Solver has two functions that must be implemented: the optimize function and the evaluate function. The evaluate function returns the agent's fitness, and the optimize function defines a single step of optimization. For example, if the agents were optimized by David Ha's SimpleES (Ha, 2017), the optimize function would first ask for potential solutions, evaluate them, return the answer to the optimizer and return the best agent found so far. However, there are other ways to define the optimization function as well. The default Single Agent Solver uses RLlib's (Liang et al., 2018) optimization algorithms (PPO, DDPG, OpenAI-ES, etc.).

Solvers are also our core unit of distribution for parallelizing work within Watts. Since OEL systems such as (Wang et al., 2019; Team et al., 2021), and (Dharna et al., 2020) optimize a large population of agent/environment pairs, we define each Solver as a Ray Actor ((Moritz et al., 2018)) so that each can optimize their models using separate cores, or even separate servers in a distributed setting. Providing a common distribution pattern allows researchers using Watts to ignore the complicated details of distributing work while keeping the performance benefits achieved through parallelization.

3.4 TRANSFER STRATEGIES

One of the bottlenecks of algorithms like POET is that the transfer-learning step contains a combinatorial explosion of work to estimate a statistical ranking problem (i.e. what is the best agent for a given environment?). For increased code-modularity and re-use, Watts breaks down the ranking problem into two linked strategies: *Scoring* and *Ranking*. Scoring and ranking are separated to allow for careful study of how structuring the interaction (e.g., different types of ranking schemes) of individuals within a population affects learning (Jiang et al., 2009; Garnelo et al., 2021). Separating ranking and scoring allows us to define what best means in multiple ways. For example, we could select the agent that scores the highest on a given task, or we could measure which environment induced the highest amount of information gain to pick a "best" agent even if it did not get the highest reward.

So far in POET-style algorithms, two ranking algorithms that have been considered. First is scoring the agents in zero-shot evaluations on every environment and selecting the single best agent for each environment (Dharna et al., 2020). A zero-shot evaluation scores an agent on a new environment without any additional optimization. The second ranking algorithm explored in POET is scoring zero-and-one-shot evaluations and selecting the single best agent for each environment (Wang et al., 2019).

For example, a zero-shot scoring strategy takes a Cartesian product of the current agents and environments returning a score for each agent in each environment (all evaluated in parallel). Then the rank strategy accepts the score matrix and takes an $argmax$ over the columns to pick new weights for each agent and environment pair. That is, the new weights for environment k are picked via: $\arg\max_\theta([R(\theta_1, E_k), ..., R(\theta_m, E_k)])$ where $R(\theta_m, E_k)$ is the zero-shot score of the mth agent on environment k.

Similarly, to implement the transfer strategy explored in (Wang et al., 2019), a new transfer strategy to pick weights for each agent-and-environment pair needs only wrap the zero-shot transfer strategy defined above and 1) perform a zero-shot evaluation 2) take one step of optimization for each agent 3) perform a one-shot evaluation and 4) take an argmax over the columns of the stacked evaluation matrices. That is, the new weights for environment k are picked via: $\arg\max_\theta([R(\theta_1, E_k), ..., R(\theta_m, E_k), R(\theta'_1, E_k), ..., R(\theta'_m, E_k)])$ where $R(\theta'_m, E_k)$ is the score of the mth agent after one-step of optimization evaluated on the kth environment.

4 PUTTING THE PIECES TOGETHER

Figure 1 shows an example of the *Manager* class in Watts, which stores a population of agent-environment pairs. While the Manager class is itself algorithm agnostic, POET is diagrammed for the purpose of illustration. Implementation details for any given algorithm are specified in the component abstractions.

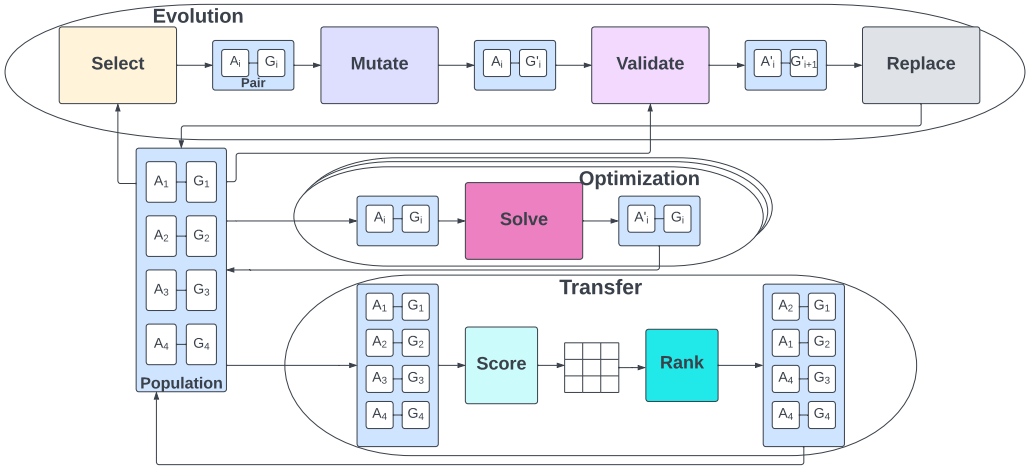


Figure 1: **Watts: The (POET)Manager Class** The POETManager stores an archive of agent and environment-generator pairs managed by Evolution, Optimization, and Transfer Components. Each of these components are further atomized and can be recombined to fit many different algorithms.

4.1 DIAGRAMMING POET

The *POETManager* is responsible for managing the inner-outer meta-learning loop of POET through components shown in Figure 1. The box of agent-environment pairs on the left is an archive of agent-environment pairs that represent the active population. The Evolution unit mutates the distribution of learning environments through a minimum criterion (MC)-based Evolution Strategy outlined as Evolution at the top of Figure 1. This collection of evolutionary components defines how to select parents from the meta-population, that are then evaluated by the Optimization components in the middle (i.e., a Solver). The Solver is where the algorithm for inner-loop learning is specified. The Transfer components at the bottom of Figure 1 scores, ranks, and selects the best agents for a given environment (pseudocode shown in Appendix A.1). So long as newly defined strategies (evolution, transfer, or optimization) follow their required APIs, new strategies can be slotted into the POETManager to experiment with creating new POET-like algorithms.

4.2 PAIRED

While designed initially for POET-like algorithms, Watts also supports multi-agent reinforcement learning (MARL) (Dennis et al., 2020) for open-ended learning. Rather than keeping an archive of agent-environment pairs, PAIRED (Dennis et al., 2020) trains a triple of agents to both build and solve 2-D mazes from the MiniGrid suite (Chevalier-Boisvert et al., 2018). Agent training is instead framed as an adversarial competition between a protagonist agent who wants to solve a maze and its asymmetric antagonist duo, which is an agent-environment pair. This adversarial duo includes a Generator artificial neural network (ANN), that takes feedback from its adversarial agent about the quality of the level that it has created. Together the goal is that the generator creates solvable levels as indicated by the performance of the antagonist agent, but that the protagonist will struggle to solve (Dennis et al., 2020) shown in Figure 2. Formally, optimizing the minimax regret is performed by the adversary game-builder and the antagonist game-player who maximize the positive regret while the protagonist maximizes negative regret. For convenience, since the adversary and antagonist networks share a loss function, they will be denoted θ_A and the protagonist agent will be denoted θ_P. The joint optimization of all three networks will continue until the networks arrive at a Nash Equilibrium – a point in policy-space where no player has anything to gain by changing only their own strategy.

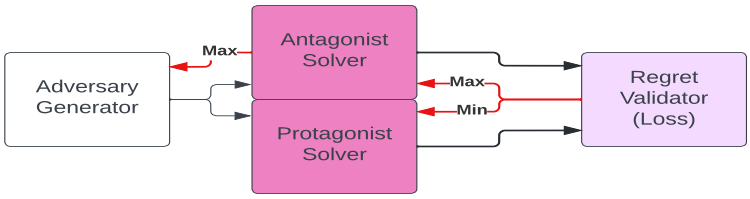


Figure 2: A potential diagram of PAIRED in Watts. In PAIRED, the adversary, antagonist, and protagonist are all parameterized by trainable neural networks and will require Validators and Generators to be differentiable objects. Black arrows are forward propagation and red arrows denote backwards gradient flow.

4.3 Unifying POET & PAIRED

Like any stochastic based population training, the active population in POET is controlled by evolutionary mechanics. Specifically in POET parents are selected randomly from the archive and removed based on their age in the archive. The MC filtering evolves the population according to the following update rule: WLOG, select an environment, E_i to mutate,

$$E_{\text{child}} \sim E_i$$
$$MC := MC_{min} \leq R(\theta_{parent}, \mathrm{E}_{child}) \leq MC_{max}$$
$$E_{i+1} \leftarrow \mathbb{1}(MC).$$

Because the MC is an indicator function, it can only impart minimal geometric knowledge regarding how to move in the latent search space. However, PAIRED optimizes following the gradients of their formulation of the MC, where loss is calculated as the minimax regret between the antagonist and protagonist agents. Let E_{θ_A} be the environment created by the adversary. Then a regret-based MC is defined as:

$$MC := R(\theta_A, E_{\theta_A}) - R(\theta_P, E_{\theta_A})$$

It follows then that agents following the loss function in PAIRED are following the directional gradients of the MC:

$$\arg\min_{\theta_P} \arg\max_{\theta_A} (\nabla MC)$$

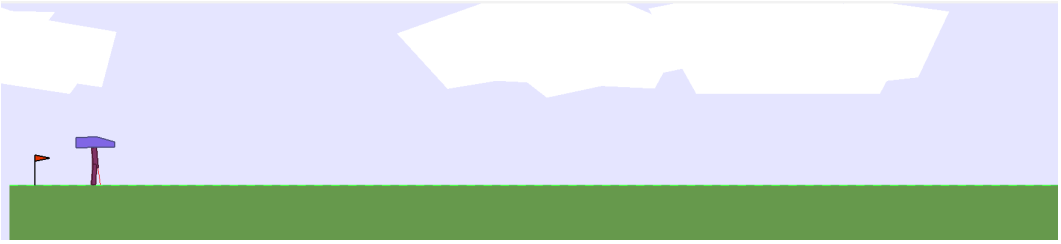
where θ_A seeks to maximize the regret signal while θ_P seeks to minimize the regret signal. Such formulation of the MC, while antithetical to the original formulation of an MC, may provide inspiration for new OEL designs and incorporating regret-based MCs into POET.

5 Comparative Study

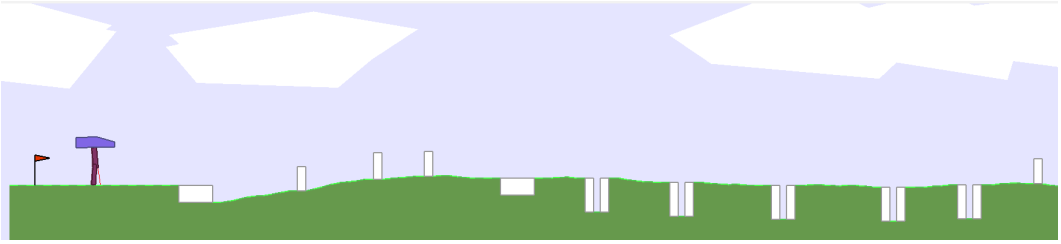
To show how Watts can incorporate different algorithms, a POETManager runs the POET algorithm on the HardcoreBipedalWalker environment and the PINSKY algorithm on a port of the 2-D maze environment from the MiniGrid suite (Chevalier-Boisvert et al., 2018) using the same Manager. Similar to the MiniGrid suite mazes, the maze environment shown is written in Griddly. The PAIRED algorithm is implemented in Watts and runs on the same port.

5.1 Original POET in Walker and Maze

The first experiment verifies that the POET-in-walker implementation is correct. As much of the walker-specific POET code is adapted from (Wang et al., 2019), it is unsurprising that the population behavior and generated terrains (Figure 3) resemble those reported in (Wang et al., 2019). Interestingly when the terrain in the HardcoreBipedal walking environment was flat, a deterministic policy trained with the OpenAI-ES algorithm quickly converged to weights that resulted in rewards over the MC_{max} threshold. However, the same policy did not score well when transferring to more complicated levels, struggling to perform better than MC_{min}. Because the agent was rewarded without learning in its environment, the algorithm stopped challenging the agent with difficult levels



(a) Initial flat terrain that seeds POET



(b) Example POET Generated Level

Figure 3: **Controllable Hardcore Bipedal Walker Environments integrated in Watts**. Bipedal agents in the Hardcore Bipedal Walker Environments start on the left hand side of the screen indicated by a flag shown in a) and b). While the terrain in a) is flat, the first obstacle the agent in b) encounters is a stair that it must step on or over to continue walking. The agent must then navigate over three stumps and a stair before encountering five pits and a stump at the rightmost portion of the environment.

it would normally add to the active population. However, training stochastic policies for the walker restored the expected behavior.

A useful experiment that can be run with Watts is to validate the assumptions made in (Dharna et al., 2020). When translating POET to work in the space of grid-worlds the authors made assumptions about how the sparsity of rewards in "game"-style settings – where agents are only rewarded for completing specific tasks (e.g. picking up a key, killing a monster) – would be too sparse to allow for the MC first explored in POET to work, as the MC requires a well-aligned and reward-dense signal (Dharna et al., 2020). Therefore, by only changing the game environment and neural network structure, Watts can run the experiment of "what would happen if $MC_{min} \leq R(\theta_{parent}, \mathrm{E}_{child}) \leq MC_{max}$ from POET were used in the maze domain"? In the Maze domain, if the agent reaches the goal, the agent receives a reward of $R = 1 - (M/T)$, where M is the timestep on which the agent found the goal and T is the timestep horizon (500). In POET, the MC_{min} and MC_{max} are hyperparameters denoting an acceptable range of reward defining if levels are too-easy and too-hard. After 10,000 loops, Maze-based POET with $MC_{min} = 0.1$ and $MC_{max} = 0.9$ had a successful reproduction rate of $\frac{1}{800}$ with virtually all of the proposed levels being rejected due to being "too hard" because the agent scores a reward of less than 0.1 – validating the assumptions. Such strict criteria (in the space of sparse-rewarding games) filtered out all chances for the coevolution to bootstrap itself as desired.

5.2 PAIRED IN MAZE

Watts seeks to remedy the fact that until now attempts to compare POET against MARL algorithms have not been possible due to the high coupling of POET to the walker environment. To start this process, Watts provides environment wrappers that when combined with multi-agent learning algorithms in RLlib recreates the PAIRED algorithm defined in (Dennis et al., 2020) for *any* single-agent game that is defined in Griddly.

As implemented in (Dennis et al., 2020), the adversary is constrained such that, at every time step, it chooses where to place a prespecified sequence of objects at available (x, y) locations. Relaxing the sequencing constraint by altering the Generator requires the adversary to also decide whether to

place an object and if so to decide the type (i.e., "[place/not-place] [object type] at ([x], [y])"). With this loosened constraint and assuming that PAIRED converges to the Nash Equilibrium (Dennis et al., 2020), its learned behavior is characterized as a blank level without a goal, agent, or maze structure. Without these structural components, all agents receive zero reward.

6 PERFORMANCE

While performance is particularly important in OEL frameworks given the large amount of computational resources necessary to run experiments, often a significant portion of the distributed communication and execution code is rewritten when implementing algorithms for reinforcement learning (Liang et al., 2018). For example, Dharna et al. (2020) implement custom code for distributed computing to run experiments on their POET-like algorithm, PINSKY (Dharna et al., 2020). Written in Python, experiments in grid-world environments completed a total of 5000 iterations in $\sim$2 weeks. On the other hand, the same experiments ran more quickly in Watts, with a total of 10000 iterations in $\sim$24 hours. Whether it is run locally or remotely on a cluster, computation in Watts is distributed through the Ray library (Moritz et al., 2018). Well-reasoned modules can promote comparison, and Watts implements some of the patterns common in distributed computing to ease implementing new OEL algorithms.

Interestingly, optimizing with the Open-AI ES (Salimans et al., 2017) algorithm Watts runs about 10000 iterations in $\sim$3.5 days on 18 CPU cores. Optimizing the same HardcoreBipedalWalker environment, Wang et al. (2019) report 60,000 iterations in 12 days (Wang et al., 2019; 2020)) on 750 CPU cores. Shown in Table 1, when Watts optimizes agent performance with PPO (Schulman et al., 2017) it completes a total of 10000 POET iterations in $\sim$28 hours. It is hypothesized that Watts' ES POET runs more slowly than the original POET due to the number of compute cores the original algorithm used and additional experiments scaling up Watts are needed to test this.

Algorithm	Implementation	Environment	$\frac{\text{loops}}{\text{cpu hr}}$
Pinsky	Original	GVGAI	0.47
Pinsky	Watts	Griddly	26.06
ES POET	Original	Walker	0.28
ES POET	Watts	Walker	6.61
PPO POET	Watts	Walker	22.31

Table 1: Algorithm runtime comparisons.

7 DISCUSSION AND FUTURE WORK

The experiments show that we fulfilled our original design goal: an atomized framework for OEL algorithms such that only modular substitutions are needed to implement different algorithms. In Watts, POET and PINSKY only differ in how three modules are implemented. The Validator in POET requires that an agent performs within a given range of fitness on a candidate level. Whereas in PINSKY, the Validator checks for the existence of solutions to the level. For the Generator object, the environment encoding is changed based on the experimental domain. For PINSKY, the Ranking object performs zero-shot evaluations on the existing levels in the active population. Furthermore, the same POETManager can run on different domains with only changes to the Generator and representation of the agent in the Solver. In both cases only small implementation changes are required to run the algorithms in Watts. The goal of Watts is to explore the space of OEL algorithms, and Watts is a simple, flexible, and composable library.

A consequence of modularity is illustrated in the head-to-head comparison of POET and PAIRED in Section 5. We provide a hypothetical Manager to implement the recent ACCEL algorithm Parker-Holder et al. (2022) in Figure 4. To improve the framework and its contribution to the community, we plan to add components for more OEL algorithms (Jiang et al., 2021b; Parker-Holder et al., 2022; Bontrager & Togelius, 2021; Team et al., 2021). Interestingly, even just thinking about running PAIRED in the HardcoreBipedalWalker environment inspires the notion of differentiable Generators

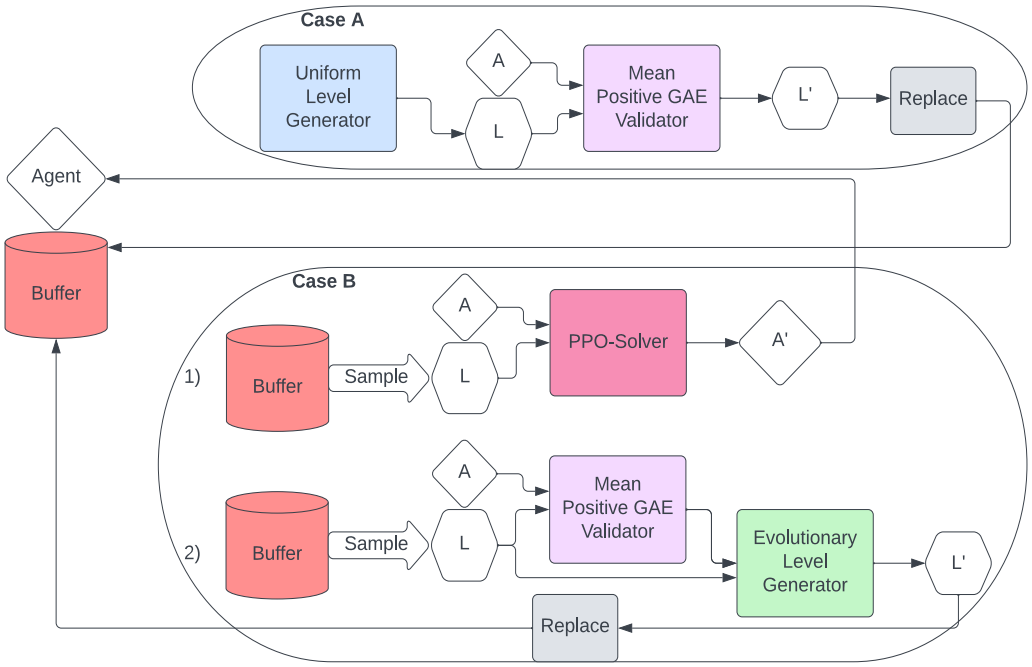


Figure 4: A hypothetical ACCELManager implemented out of Watts modules. For any given loop, sample case A or case B from a Bernoulli Random Variable. Case B is a two step process with one optimization step and one evolutionary-MC step. Note each individual component has been created just not strung together yet.

and Validators. Furthermore, something like a Generator API could provide more functionality, like the ability to sequentially build levels. With more functionality, the aim is to build an active core community of people interested in all aspects of OEL (including procedurally generated content).

Going beyond the comparison of existing algorithms, Watts can potentially enable the invention of new and more capable open-ended learning methods. One aim of Watts is to enable the easy testing of new algorithmic ideas through its modularity, like the recent population-based extension to PAIRED (Du et al., 2022) that pushes the algorithm even closer to POET. Pushing POET more toward PAIRED, we aim to explore incorporating a multi-agent minimax regret signal like that in PAIRED. Both innovations to these algorithms are conceivable in the Watts framework.

Finally, the goal of Watts is to explore the space of OEL algorithms therefore designing simple, flexible, and composable algorithmic interfaces are a must; in order for others to use Watts, we must have documentation instructing how to setup and easily use the system. It must be able to be run on a variety of operating systems without performance disparity. And there must be a community of active users and contributors guiding and encouraging any newcomers.

8 Conclusions

We presented Watts, a framework for comparing existing open-ended learning methods and inventing new ones. In the experiments described here, we implemented the well-known POET and PAIRED algorithms and compared them, showing that they are each dependent on the particular problem domain they are applied to. We believe that breaking existing methods into their constituent components and recomposing them in a common framework will enable us and others to more rapidly iterate on novel OEL methods.

REFERENCES

Chris Bamford, Shengyi Huang, and Simon Lucas. Griddly: A platform for ai research in games, 2020.

Margaret A. Boden. Computer models of creativity. *AI Magazine*, 30(3):23, July 2009. doi: 10.1609/aimag.v30i3.2254. URL https://doi.org/10.1609/aimag.v30i3.2254.

Philip Bontrager and Julian Togelius. Learning to generate levels from nothing. In *2021 IEEE Conference on Games (CoG)*, pp. 1–8. IEEE, 2021.

Matthew Botvinick, Sam Ritter, Jane X. Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement learning, fast and slow. *Trends in Cognitive Sciences*, 23(5):408–422, May 2019. doi: 10.1016/j.tics.2019.02.006. URL https://doi.org/10.1016/j.tics.2019.02.006.

Jonathan C Brant and Kenneth O Stanley. Minimal criterion coevolution: a new approach to open-ended search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 67–74, 2017.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. https://github.com/maximecb/gym-minigrid, 2018.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design, 2020.

Aaron Dharna, Julian Togelius, and L. B. Soros. Co-generation of game levels and game-playing agents. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1):203–209, Oct. 2020. URL https://ojs.aaai.org/index.php/AIIDE/article/view/7431.

Yuqing Du, Pieter Abbeel, and Aditya Grover. It takes four to tango: Multiagent selfplay for automatic curriculum generation, 2022.

Marta Garnelo, Wojciech Marian Czarnecki, Siqi Liu, Dhruva Tirumala, Junhyuk Oh, Gauthier Gidel, Hado van Hasselt, and David Balduzzi. Pick your battles: Interaction graphs as population-level objectives for strategic diversity, 2021.

Maxim Gumin. Wave Function Collapse Algorithm, 9 2016. URL https://github.com/mxgmn/WaveFunctionCollapse.

David Ha. Evolving stable strategies. *blog.otoro.net*, 2017. URL http://blog.otoro.net/2017/11/12/evolving-stable-strategies/.

Nikolaus Hansen. The cma evolution strategy: A comparing review. *Towards a new evolutionary computation. Studies in Fuzziness and Soft Computing*, 192:75–102, 06 2007. doi: 10.1007/3-540-32494-1_4.

Minqi Jiang, Michael Dennis, Jack Parker-Holder, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Replay-guided adversarial environment design, 2021a.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay, 2021b.

Xiaoye Jiang, Lek-Heng Lim, Yuan Yao, and Yinyu Ye. Statistical ranking and combinatorial hodge theory, 2009.

Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. In *Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'20. AAAI Press, 2020. ISBN 978-1-57735-849-7.

Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 3053–3062. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/liang18b.html.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 561–577, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktaschel. Evolving curricula with regret-based environment design, 2022. URL https://arxiv.org/abs/2203.01302.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Bob Ross. *Happy Little Accidents: The Wit & Wisdom of Bob Ross*. Running Press Adult, 2017.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017. URL https://arxiv.org/abs/1703.03864.

Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in Psychology*, 4, 2013. ISSN 1664-1078. doi: 10.3389/fpsyg.2013.00313. URL https://www.frontiersin.org/article/10.3389/fpsyg.2013.00313.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

L. B. Soros, Nick Cheney, and Kenneth O. Stanley. How the strictness of the minimal criterion impacts open-ended evolution. In *Proceedings of the Artificial Life Conference 2016*. MIT Press, 2016.

Lisa Soros. *Necessary Conditions for Open-Ended Evolution*. PhD thesis, University of Central Florida, 08 2018.

RUSSELL K. STANDISH. OPEN-ENDED ARTIFICIAL EVOLUTION. *International Journal of Computational Intelligence and Applications*, 03(02):167–175, June 2003. doi: 10.1142/s1469026803000914. URL https://doi.org/10.1142/s1469026803000914.

Kenneth O. Stanley. Art in the sciences of the artificial. *Leonardo*, 51(2):165–172, 2018.

Open Ended Learning Team, Adam Stooke, Anuj Mahajan, Catarina Barros, Charlie Deck, Jakob Bauer, Jakub Sygnowski, Maja Trebacz, Max Jaderberg, Michael Mathieu, Nat McAleese, Nathalie Bradley-Schmieg, Nathaniel Wong, Nicolas Porcel, Roberta Raileanu, Steph Hughes-Fitt, Valentin Dalibard, and Wojciech Marian Czarnecki. Open-ended learning leads to generally capable agents, 2021. URL https://arxiv.org/abs/2107.12808/.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. POET: Open-ended coevolution of environments and their optimized solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pp. 142–151. ACM, 2019.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeffrey Clune, and Kenneth O. Stanley. Enhanced POET: open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pp. 9940–9951. PMLR, 2020.

A Appendix

A.1 Pseudocode

```
class BirthThenKillStrategy(EvolutionStrategy):
    def __init__(
            self,
            level_validator: LevelValidator,
            replacement_strategy: ReplacementStrategy,
            selection_strategy: SelectionStrategy,
            mutation_rate: float = 0.8,
    ):
        self._level_validator = level_validator
        self._replacement_strategy = replacement_strategy
        self._selection_strategy = selection_strategy
        self._mutation_rate = mutation_rate

    def evolve(self, active_population: list, birth_func: Callable[[List[Tuple]], List]) -> list:
        children = []
        potential_parents = self._selection_strategy.select(active_population)

        for parent in potential_parents:
            new_generator = parent.generator.mutate(self._mutation_rate)
            is_valid, data = self._level_validator.validate_level(generators=[new_generator],
                                                                  solvers=[parent.solver])
            if is_valid:
                children.append((parent.solver, new_generator, parent.id, 0))
        children = birth_func(children)
        active_population.extend(children)
        return self._replacement_strategy.update(active_population)
```

Figure 5: Evolution Strategy Pseudocode for an MC-guided Outer-Loop

```python
@ray.remote
class Solver:
    def __init__(self, trainer, agent, env, ...):
        self.trainer = trainer(agent, env)
        self.agent = agent
        self.env = env

    def optimize(self):
        metrics = self.trainer.train(self.agent, self.env)
        return metrics

    ... other helper functions
```

Figure 6: Example Solver that implements an optimization algorithm (trainer) on a remote process.

```python
class GetBestSolver(RankStrategy):
    def __init__(self, scorer):
        self.scorer = scorer

    def transfer(self, solver_list, generator_list) -> dict:
        """Run the transfer tournament; take in a solver list and a generator list.
        :param solver_list:
        :param generator_list:
        :param id_map:
        :return: dict of new weights indexed by a pair_id.
        """
        self.t += 1

        solvers, solver_idxs = zip(*solver_list)
        generators, generator_idxs = zip(*generator_list)
        solver_generator_combo_id = list(product(solver_idxs, generator_idxs))
        tournament_results = self.scorer.score(solvers=solvers,
                                               generators=generators,
                                               id_map=solver_generator_combo_id)


        self.tournaments[self.t] = tournament_results
        generator_id_matrix = tournament_results[..., 0]
        solver_id_matrix = tournament_results[..., 1]
        best_indicies = np.argmax(tournament_results[..., 2], axis=0)
        # get the new weights
        new_weights = {}
        for i, g_id in enumerate(generator_idxs):
            for j, s_id in enumerate(solver_idxs):
                best_solver_id = solver_id_matrix.T[i, best_indicies[i]].item()
                if s_id == best_solver_id:
                    new_weights[g_id] = (
                        ray.get(solvers[j].get_weights.remote()),
                        best_solver_id
                    )

        return new_weights
```

Figure 7: Transfer Strategy Pseudocode: Picking the best zero-shot agent for each env

```python
class POETManager:
    def __init__(self,
            exp_name: str,
            initial_pair,
            evolution_strategy: EvolutionStrategy,
            transfer_strategy: RankStrategy):

        self._evolution_strategy = evolution_strategy
        self._transfer_strategy = transfer_strategy
        self.active_population = [initial_pair]

    def run(self):
        while self.i <= self.args.num_poet_loops:

            if self.i % self.args.evolution_timer == 0:
                self.active_population = self._evolution_strategy.evolve(
                    active_population=self.active_population,
                    birth_func=self.build_children # how to instanciate new POET-PAIRS
                )
            pop_opt_results = self.optimize_each_pair()

            if self.i % self.args.transfer_timer == 0:
                nets = [(p.solver, p.id) for j, p in enumerate(self.active_population)]
                lvls = [(p.generator, p.id) for j, p in enumerate(self.active_population)]
                new_weights = self._transfer_strategy.transfer(nets, lvls)

                for j, (best_w, best_id) in new_weights.items():
                    self.active_population[j].update_solver_weights(best_w)

            self.i += 1
```

Figure 8: The POET algorithm modular form