Syllabus: Portable Curricula for Reinforcement Learning Agents

Ryan Sullivan, Ryan Pégoud, Ameen Ur Rehman, Xinchen Yang, Junyun Huang, Aayush Verma, Nistha Mitra, John P. Dickerson

Keywords: Curriculum Learning, Unsupervised Environment Design, Open-Endedness

Summary

Curriculum learning has been a quiet, yet crucial component of many high-profile successes of reinforcement learning. Despite this, it is still a niche topic that is not directly supported by any of the major reinforcement learning libraries. These methods can improve the capabilities and generalization of RL agents, but often require complex changes to training code, limiting their impact on the field. We introduce Syllabus, a portable curriculum learning library, as a solution to this problem. Syllabus provides a universal API for curriculum learning, modular implementations of popular automatic curriculum learning methods, and infrastructure that allows them to be easily integrated with asynchronous training code in nearly any RL library. Syllabus provides a minimal API for core curriculum learning components, making it easier to design new algorithms and adapt existing ones to new environments. We demonstrate this by evaluating the algorithms in Syllabus on several new environments, each using agents written in a different RL library. We present the first examples of automatic curriculum learning in NetHack and Neural MMO, two of the most challenging RL benchmarks, and find evidence that existing methods do not easily transfer to complex new environments.

Contribution(s)

1. This paper introduces Syllabus, a library of portable curriculum learning algorithms and infrastructure for synchronizing curricula across reinforcement learning environments running in separate processes. Syllabus includes portable implementations of several popular automatic curriculum learning algorithms and tools for manually designing curricula.
   Context: There are open-source curriculum learning libraries (Jiang et al., 2022; 2023; Dharna et al., 2022; Coward et al., 2024), but they build curriculum logic into the RL training code, making it difficult to extend methods and apply them to new environments. Syllabus is the first portable infrastructure for curriculum learning.
2. We evaluate tuned curriculum learning baselines in 4 environments including 2 which have not previously been explored in the context of curriculum learning. These baselines provide a solid foundation for future curriculum learning research.
   Context: We implement baselines from Jiang et al. (2021b), Kanitscheider et al. (2021), Zhang et al. (2023), and Rutherford et al. (2024) and apply all four algorithms to baseline environments used in these works. Some of these evaluations are reproductions of the experiments in those papers, but most are novel.
3. Our experiments demonstrate that popular curriculum learning methods are far less effective outside of the environments in which they were originally developed, and that more advanced methods may be necessary in complex environments.
   Context: Previous work successfully used automatic curricula over level seeds to train agents in procedurally generated environments. We show that these curricula over level seeds are ineffective in new or complex environments.

Syllabus: Portable Curricula for Reinforcement Learning Agents

Ryan Sullivan [1], Ryan Pégoud [2], Ameen Ur Rehman [3], Xinchen Yang [1], Junyun Huang [1], Aayush Verma [1], Nistha Mitra [1], John P. Dickerson [1]
rsulli@umd.edu

[1]**University of Maryland, College Park** [2]**University College London**
[3]**Jamia Hamdard University**

Abstract

Curriculum learning has been a quiet, yet crucial component of many high-profile successes of reinforcement learning. Despite this, it is still a niche topic that is not directly supported by any of the major reinforcement learning libraries. These methods can improve the capabilities and generalization of RL agents, but often require complex changes to training code. We introduce Syllabus, a portable curriculum learning library, as a solution to this problem. Syllabus provides a universal API for curriculum learning, modular implementations of popular automatic curriculum learning methods, and infrastructure that allows them to be easily integrated with asynchronous training code in nearly any RL library. Syllabus provides a minimal API for core curriculum learning components, making it easier to design new algorithms and adapt existing ones to new environments. We demonstrate this by evaluating the algorithms in Syllabus on several new environments, each using agents written in a different RL library. We present the first examples of automatic curriculum learning in NetHack and Neural MMO, two of the most challenging RL benchmarks, and find evidence that existing methods do not directly transfer to complex new environments. Syllabus can be found at https://github.com/RyanNavillus/Syllabus.

1 Introduction

Curricula have been a core component of many of the successes of reinforcement learning (RL). AlphaGo (Silver et al., 2016) was trained with self-play, AlphaStar used a novel league training method to achieve grandmaster level play in Starcraft II (Vinyals et al., 2019), and GT Sophy (Wurman et al., 2022) was taught to outrace professionals in Gran Turismo with manually curated sections of race tracks. Curriculum learning is essential in environments with large task spaces where many tasks are too challenging or too easy to provide a useful learning signal. In these settings agents must prioritize tasks that teach transferable skills to accelerate learning in new tasks. This problem is especially pronounced in open-ended environments with infinite or evolving task spaces, similar to the real world. Open-endedness research seeks to co-evolve agents and environments in order to create more complex tasks and incentivize more intelligent agent behavior (Wang et al., 2019; 2020). To approximate this complexity, the field focuses on the most challenging RL benchmarks including Minecraft (Guss et al., 2019; Fan et al., 2022), NetHack (Küttler et al., 2020), and Neural MMO (Suarez et al., 2019; Rosseau et al., 2022; Suarez et al., 2024). Curriculum learning is an integral component of open-ended processes, so they directly benefit from curriculum learning research.

Curriculum learning (CL) methods fit naturally into the RL framework by modifying the data distribution that an agent experiences. According to Narvekar et al. (2020), curriculum learning explores how tasks or data samples can be sequenced to learn problems that can not be solved directly. Au-

tomatic curriculum Learning (ACL) methods (Graves et al., 2017; Portelas et al., 2020b) focus on autonomously sequencing these tasks to maximize agent performance, and have been shown to outperform random task ordering in robotics (Tobin et al., 2017; OpenAI et al., 2019; Mehta et al., 2020), videogames (Salimans & Chen, 2018; Berner et al., 2019), and navigation (Florensa et al., 2018; Racaniere et al., 2019; Portelas et al., 2020a). Despite the near ubiquity of curricula in successful applications of reinforcement learning, there is little support for these methods in standard RL infrastructure. In theory, CL provides orthogonal benefits to policy optimization algorithms and they can be integrated with minimal restrictions. In practice however, CL methods are diverse, complex, and necessitate significant changes to training code. CL algorithms can modify the environment configuration (Dennis et al., 2020; Jiang et al., 2021b;a), introduce new neural networks to train (Ostrovski et al., 2017; Pathak et al., 2017a; Bellemare et al., 2016b; Henaff et al., 2022), or even query tasks from a large language models (Colas et al., 2023; Du et al., 2023b; Zhang et al., 2023; Faldor et al., 2024). Each method is different, so engineering time often scales linearly with the number of methods being evaluated.

Most open-source implementations of CL algorithms integrate their code directly within the RL algorithm, intertwining CL and RL logic. We argue that this is typically done to accommodate existing multiprocessing implementations. ACL methods maintain a sampling distribution over tasks which they update using feedback from the environment. In an asynchronous RL setting, this requires additional data exchanges between environments running on separate processes and the curriculum in the main process. The simplest solution is to add these extra messages to the existing message-passing infrastructure, usually in the `infos` dictionary. However, accessing this data requires direct changes to the training code that vary depending on the CL method, so CL algorithms are implemented as standalone libraries with custom training code. This entanglement of CL and RL code makes it difficult to isolate algorithmic details and apply CL methods beyond their original codebases, ultimately limiting reproducibility and hindering future research.

Syllabus addresses this problem by introducing a simple, portable approach to designing curriculum learning algorithms. We implement CL algorithms as modular additions to RL code, complementing their theoretical orthogonality to policy optimization. Syllabus makes minimal assumptions about the RL training code and establishes a separate synchronization pathway between the curriculum and asynchronous environments. This architecture best enables future research by integrating with existing RL infrastructure rather than attempting to replace it. We demonstrate the generality of Syllabus by implementing multiple ACL algorithms and evaluate them in several RL libraries including CleanRL (Huang et al., 2022), RLLib (Liang et al., 2018), moolib (Mella et al., 2022), and PufferLib (Suarez, 2023). This allows us to present new baselines of several ACL algorithms on Procgen (Cobbe et al., 2020b), and Crafter (Hafner, 2021), and the first ACL results on NetHack (Küttler et al., 2020), and Neural MMO (Suarez et al., 2019; 2024).

2 Background

Curriculum Learning has been studied in the context of deep supervised learning for many years (Bengio et al., 2009; Elman, 1993). More recently, it has been used to improve the capabilities and generalization of deep reinforcement learning agents. Curriculum learning encompasses a wide range of methods that change the training data distribution. The goal is to increase the asymptotic performance or sample efficiency of RL agents on a single environment or range of tasks by sampling tasks that provide maximal learning value. These methods often make a distinction between the goal generating teacher and the student agent that plays the tasks assigned by the teacher. Narvekar et al. (2020) and Portelas et al. (2020b) present more thorough taxonomies and surveys of existing curriculum learning methods. It can also be viewed as an extension of transfer learning, which Taylor & Stone (2009) and Zhu et al. (2023) summarize in the context of RL.

Many diverse methods fall under the broad definition of curriculum learning. Some take inspiration from the Zone of Proximal Development (Vygotsky, 1978; Chaiklin, 2003) which suggests that tasks in the proximal zone – tasks that are neither too hard nor too easy – maximize learning

progress (LP). Many curriculum learning papers therefore focus on developing measures of learning progress. Portelas et al. (2020a) fit Gaussian Mixture Models (Rasmussen, 1999) to a dataset of continuous tasks and their corresponding learning progress measures, then treat the individual Gaussians as arms in a bandit problem, allowing a teacher to bias training toward high-LP tasks. Kanitscheider et al. (2021); Tzannetos et al. (2023); Rutherford et al. (2024) evaluate the agent's current progress on the full task space throughout training, and compute a learning progress metric from these success rates. Klink et al. (2022) introduce an optimal transport based curriculum and identify a link between interpolation based curricula and methods that use success rates, regret, or learning progress. Intrinsically motivated exploration bonuses like curiosity (Pathak et al., 2017b) or novelty (Bellemare et al., 2016a; Taiga et al., 2021; Henaff et al., 2022) modify the reward function to induce a curriculum by incentivizing the agent to explore unseen section of the state space.

Self-play based methods (Samuel, 1959; Tesauro, 1995; Silver et al., 2017) create an implicit curriculum in competitive multiplayer games by training a policy against itself (Leibo et al., 2019). As the agent becomes more capable, so does the opponent, allowing it to continually improve. Training only against the current policy can lead to strategic cycles, but mixing in past policies helps prevent this (Brown, 1951; Heinrich et al., 2015b; Heinrich & Silver, 2016; Lanctot et al., 2017; Vinyals et al., 2019). Self-play has been extended to include multiple students (Sukhbaatar et al., 2018; OpenAI et al., 2021) and to train a teacher represented by a neural network (Du et al., 2023a; Dennis et al., 2020; Mediratta et al., 2023a). Zhang et al. (2024) provide an extensive survey of self-play methods in RL.

Unsupervised Environment Design (UED) is another paradigm for curriculum learning proposed by Dennis et al. (2020). They differentiate UED as a framework in which environments have unspecified configuration parameters, thereby forming an Underspecified Partially Observable Markov Decision Process (UPOMDP). The parameters generate a distribution of solvable tasks, and the goal of a UED method is to train a policy that generalizes across all possible instantiations of those variables. This approach has led to several algorithms for training agents in procedurally generated games (Dennis et al., 2020; Jiang et al., 2021b;a; Beukman et al., 2024; Mediratta et al., 2023b).

The specific policy optimization algorithm used to train agents is largely independent from the choice of curriculum. For simplicity, most curriculum learning research and all of the experiments in this paper use Proximal Policy Optimization (Schulman et al., 2017). PPO is an on-policy, policy gradient algorithm that has been successfully applied to a wide range of challenging RL environments. It uses a clipped objective to avoid large changes to the policy, which prevents collapses in performance and stabilizes learning (Schulman et al., 2015; 2017).

3 Related Work

Popular RL libraries do not include curriculum learning algorithms, but there are specialized libraries for curriculum learning. The Dual Curriculum Design library (Jiang et al., 2021a) incorporates multiple UED methods in a single repository including PLR (Jiang et al., 2021b), PAIRED (Dennis et al., 2020; Mediratta et al., 2023a), Robust PLR, REPAIRED (Jiang et al., 2021a), and ACCEL (Parker-Holder et al., 2022). Watts is another curriculum learning library focused on open-endedness with implementations of POET and PAIRED (Dharna et al., 2022). It atomizes components of the open-ended framework into modules to create novel methods by combining these components in new ways. Their work also compares these algorithms on a suite of common benchmarks. Yuan et al. (2024) introduce a library of instrinsic motivation algorithms called RLeXplore, and use their library to identify algorithmic components of PPO and SAC (Haarnoja et al., 2018) that impact the performance of popular exploration bonuses. TeachMyAgent (Romac et al., 2021) is a complete benchmarking library for CL, which includes two procedural Box2D (Brockman et al., 2016) environments, a collection of ACL methods, and several student algorithm implementations. Their work categorizes several challenges for ACL methods including task space feasibility, robustness to novel RL algorithms, and required expert knowledge. TeachMyAgent implements several older ACL methods which (Jiang et al., 2021a) showed underperform relative to UED baselines.

Recently, JAX has become a popular choice for writing fast deep learning libraries that can be run end-to-end on hardware accelerators (Bradbury et al., 2018). Minimax (Jiang et al., 2023) provides JAX-based implementation of the UED algorithms in the DCD library, leading to significantly faster training. JaxUED (Coward et al., 2024) refactors the UED algorithms in Minimax into single-file implementations for faster prototyping, inspired by CleanRL (Huang et al., 2022). Both of these libraries are intended to be used with JAX-based environments, which allow you to run the entire training loop on hardware accelerators. However, they provide only moderate performance benefits to the complex CPU-based games that motivate open-endedness research, at the cost of inconvenient code constraints. These libraries also suffer from the same lack of portability as previous curriculum learning systems – their algorithms can not be easily applied to RL code written in any other library.

Syllabus distinguishes itself from previous works by making minimal assumptions on the training code, rather than providing its own training system that intermingles RL and CL. By defining a simple, uniform API for interfacing with a curriculum, Syllabus makes it possible to add these algorithms to nearly any RL system with minimal code changes. An additional benefit of this approach is that all of the algorithmic details are contained in a single place, rather than distributed across the codebase. Syllabus also provides the only general-purpose infrastructure for synchronizing curricula across CPU-based environments in multiple processes. Curriculum learning algorithms almost exclusively interact with environments rather than policy optimization, and Syllabus takes advantage of this through a unique paradigm of portable infrastructure.

4 Design Philosophy

Syllabus aims to simplify the process of developing curriculum learning methods, combining them with RL algorithms, and applying them to challenging domains. As such, it is built to be compatible with as many different RL libraries and multiprocessing methods as possible. These goals motivate the following key points of our design philosophy:

1. Syllabus should be agnostic to the choice of reinforcement learning framework.
2. Syllabus should have general APIs that support any form of curriculum learning.
3. Integrating Syllabus into training infrastructure should require minimal code changes.
4. Code complexity should scale with the complexity of the curriculum learning algorithm.
5. Algorithm implementations should be contained in a single file.

The first point motivates many of the implementation choices in Syllabus that may seem odd in isolation. To maintain compatibility with several libraries we must honor the Gymnasium and PettingZoo environment APIs (Brockman et al., 2016; Towers et al., 2024; Terry et al., 2021) and write systems that work with many different multiprocessing solutions. For instance, Gymnasium, Stable Baselines 3 (Raffin et al., 2021), and RLLib (Liang et al., 2018) all provide their own vector environment implementations.

Syllabus's utility is tied to its ability to integrate new forms of curriculum learning. The diversity of existing CL algorithms is highlighted in section 1 and subsection 5.1. Supporting all of these methods in a single API necessitates some complexity. When one method requires a new interface, we prefer to have a heterogeneous, modular API rather than complicate the interface for all methods. For example, we define a separate Curriculum and Agent API for task-based and opponent-based curricula respectively. These components are described in subsection 5.1 and can be used separately or combined to form joint curricula over tasks and opponents. In each case, we create the smallest possible API to minimize complexity for users.

Our focus on single-file implementations is inspired by the success of CleanRL (Huang et al., 2022). CleanRL provides single-file implementations of popular RL algorithms to support fast iteration and transparency, making it easy for researchers to reliably report algorithmic details. Though end-to-end single-file training scripts are inherently non-portable, Syllabus encapsulates all CL logic for each algorithm in a single class to capture much of the same transparency and simplicity.

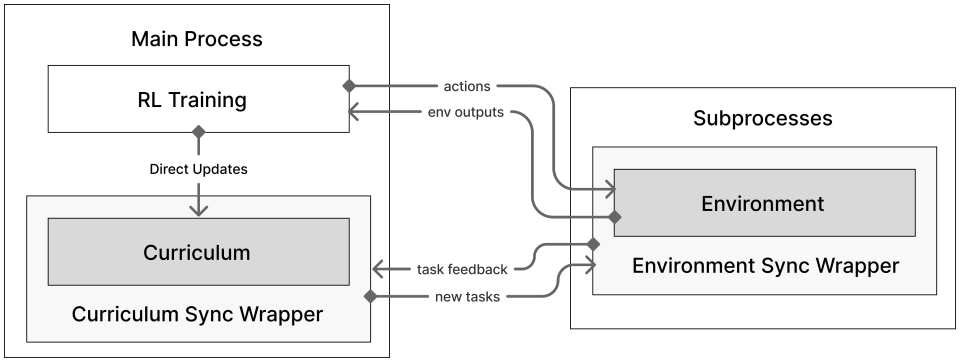


Figure 1: Syllabus with a standard asynchronous RL training setup.

5 Syllabus APIs

Syllabus designates responsibility for maintaining sampling distributions over the task space to a `Curriculum` class, and implements task swapping through a wrapper over the environment. This provides a uniform interface for setting the current task of an environment, which we explain fully in Supplementary A.3. Each environment also defines a `TaskSpace` that represents the full range of tasks that can be used to train agents. Each API is designed to be simple to use and to support future use cases.

5.1 Curriculum API

In Syllabus, a `Curriculum` is responsible for maintaining a distribution over the task space and implementing a sampling function for selecting tasks. Automatic curriculum learning methods use feedback from the RL training process to update their sampling distribution. The Curriculum API provides multiple options for incorporating information from the environments and policy, either manually or automatically through Syllabus's synchronization infrastructure. The multiprocessing approach is explained more thoroughly in subsection 5.4. The curriculum can be updated after each step, episode, or completed task with the `update_on_step`, `update_on_episode`, or `update_task_progress` methods respectively. These update options allow us to implement a diverse range of curriculum learning algorithms under the same API, as we show in subsection 5.5.

5.2 Agent API

The Agent API is a subset of the Curriculum API that defines curricula over co-players in multiagent games. These algorithms store co-players and load them at a later time based on some sampling criteria. The currently implemented algorithms are forms of self-play (SP) (Samuel, 1959; Tesauro, 1995) where the opponent is a previous copy of the online policy. These are intended for two-player competitive games, but the API supports many-agent general-sum games.

1. **Fictitious Self Play (FSP)** - maintains a history of past copies of the agent as opponents (Heinrich et al., 2015a). FSP uniformly samples an opponent from history to prevent strategic cycles.
2. **Prioritized Fictitious Self Play (PFSP)** - like FSP, PFSP maintains a history of past opponents to sample during training. PFSP selects the opponent with the highest win-rate against the current agent. This prevents the curriculum from spending a disproportionate amount of time training against opponents that the agent already performs well against.

Syllabus supports the simultaneous use of opponent-based (e.g. FSP, PFSP) and task-based (e.g. DR, PLR) curricula through the `DualCurriculumWrapper`. This wrapper extends the Curriculum API, allowing the user to sample from a joint task space and update both curricula at once. This API allows users to experiment with different joint curricula without changing their training code.

```
curriculum = DomainRandomization(task_space)
curriculum = make_multiprocessing_curriculum(curriculum)

env = NetHackScore()
env = NetHackTaskWrapper(env)
env = GymnasiumSyncWrapper(env, curriculum.components)
```

Figure 2: Using Syllabus for curriculum learning with just a few lines of code.

5.3 Task Space API

Defining a task space for training agents is a fundamental challenge in curriculum learning, similar to reward shaping for policy optimization. Task spaces are domain-specific and must be carefully designed for each environment to create an effective curricula. In most benchmark environments, tasks exist in a low-dimensional discrete or continuous space, while more complex environments may use a combination of discrete and continuous variables or intricate predicate systems, as seen in XLand (Team et al., 2023; Nikulin et al., 2023) and Neural MMO (Suarez et al., 2019). Curriculum learning implementations are often restricted to the task space representations from the environments that they were originally designed for, but Syllabus's Task API removes these limitations, making it possible to explore new task spaces without modifying algorithm code.

The Task Space API simplifies task space definition and curriculum compatibility by mapping tasks into a Gym Space (Brockman et al., 2016). This allows users to define tasks in a format suited to their environment while maintaining a simple representation within the curriculum code. For example, while Prioritized Level Replay originally used level seeds as tasks, our implementation supports any discrete task list. The PLR code handles tasks as integers for streamlined algorithm logic, while the environment interprets them as seeds, map encodings, reward functions, etc. This approach also reduces bandwidth for task transfer between processes and enables task indexing for defining separate training and validation sets.

5.4 Multiprocessing Infrastructure

The real practical challenges in curriculum learning come from synchronizing curricula using feedback from environments running in multiple processes. Syllabus's infrastructure is designed to separate curriculum and multiprocessing logic to provide interoperability with many different forms of asynchronous RL infrastructure. It uses a bidirectional sender-receiver model in which the curriculum sends tasks and the environment sends feedback from playing the provided task. The curriculum synchronization wrapper adds multiprocessing functionality to a `Curriculum` and an environment synchronization wrapper adds the same functionality to the environment without changing the Gym interface. The environment synchronization wrapper automatically sends feedback to the curriculum after each step, episode, or completed task depending on the curriculum method. You can also update the curriculum with training metrics directly from the main learner process. Figure 1 shows a diagram of how these components interconnect. The curriculum does not need immediate feedback or block training, so all updates are batched to reduce multiprocessing overhead, and task sampling is can be buffered to prevent delays at the start of each episode. Crucially, adding Syllabus's synchronization to existing RL training code requires only a few lines of code, shown in Figure 2.

The user-facing curriculum and environment code follows our design goals stated in section 4, while the multiprocessing infrastructure is engineered to ensure stability and reduce the risk of bugs. To guarantee that researchers will not need to spend time reading or debugging this code, Syllabus includes thorough integration tests, smoke tests, regression tests, and optimization benchmarks for all multiprocessing code, tested with all of the implemented curriculum learning methods. More details and performance numbers can be found in Supplementary C and Supplementary B.

5.5 Automatic Curriculum Learning Implementations

Syllabus includes portable implementations of several popular curriculum learning baselines. We prioritize modern methods which have been successfully applied to complex environments, and plan to add many more algorithms in future versions of Syllabus. We also provide utilities for manual curriculum learning or transfer learning. For example, we implement Simulated Annealing (Kirkpatrick et al., 1983), an expanding sampling range curriculum, and a sequential curriculum. The sequentual curricula trains agents on a list of individual tasks or entire curricula in stages that run until a predetermined number of steps or user-defined stopping conditions are met. Supporting all of these diverse methods helps to demonstrate the generality of Syllabus's interfaces.

Prioritized Level Replay (PLR) - a popular UED method which maintains a buffer of levels with high learning potential (Jiang et al., 2021b). PLR typically prioritizes tasks with a high value loss, though recent work explores other options (Jiang et al., 2021a; Jackson et al., 2024). It also tracks the staleness of each task's value loss score, and assigns some probability to very stale tasks.

We generalize this implementation to support arbitrary task spaces instead of only environment seeds, and implement an asynchronous version that can be updated with arbitrary batch sizes. We also implement Robust PLR (Jiang et al., 2021a) as an initialization option for PLR. PLR is a core component of several more recent UED methods (Parker-Holder et al., 2022; Samvelyan et al., 2022) and has been successfully applied to complex domains (Team et al., 2023). We sample tasks based on the L1 value loss as defined below, where λ and γ are the GAE and MDP discount factors respectively, and δt is the TD-error at timestep t:

$$\frac{1}{T}\sum_{t=0}^{T}\left|\sum_{k=t}^{T}(\gamma\lambda)^{k-t}\delta_k\right| \tag{1}$$

Learning Progress (LP) - this method was developed by Kanitscheider et al. (2021) to train PPO agents in Minecraft. Using this algorithm and an exploration bonus, their agents were able to consistently acquire diamonds, drastically surpassing PPO alone. They define a set of pass/fail tasks for the agent to achieve, and prioritize tasks with a recent change in success rate, which they call learning progress. This method tracks a fast and slow exponential moving average (EMA) of the agent's success rate for each task, which is evaluated periodically throughout training. Learning progress is calculated by taking the absolute difference between these EMA values and performing several normalization steps. High learning progress indicates a task that the agent is beginning to learn or forget, so those tasks are prioritized during sampling.

Open-endedness via Models of human Notions of Interestingness (OMNI) - this is an extension to the Learning Progress curriculum introduced by (Zhang et al., 2023) which asks an LLM to identify "interesting" tasks given the agent's current success rates on each task. The LLM filters out tasks that are uninteresting given high proficiency at another task. Starting with the highest success rate, the LLM partitions tasks into interesting and uninteresting sets, and masks uninteresting tasks out of the sampling distribution generated by the LP curriculum. This approach has recently been extended to allow the LLM to generate new interesting tasks defined in code (Faldor et al., 2024).

Sampling for Learnability (SFL) - this method is a hybrid of the approaches used by PLR and LP. It uses a simple metric for prioritizing tasks by sampling according to $p(1-p)$ where p is the task success rate, a metric called learnability that was originally described in (Tzannetos et al., 2023). The method as introduced by Rutherford et al. (2024) samples from a mixture of a uniform distribution over the top k most learnable tasks and a uniform distribution over all tasks. The mixing ratio of these distributions is controlled by a hyperparameter p. We additionally implement a version of this method that samples directly from the full distribution generated by $p(1-p)$. We find that it sometimes outperforms SFL while removing two environment-dependent hyperparameters.

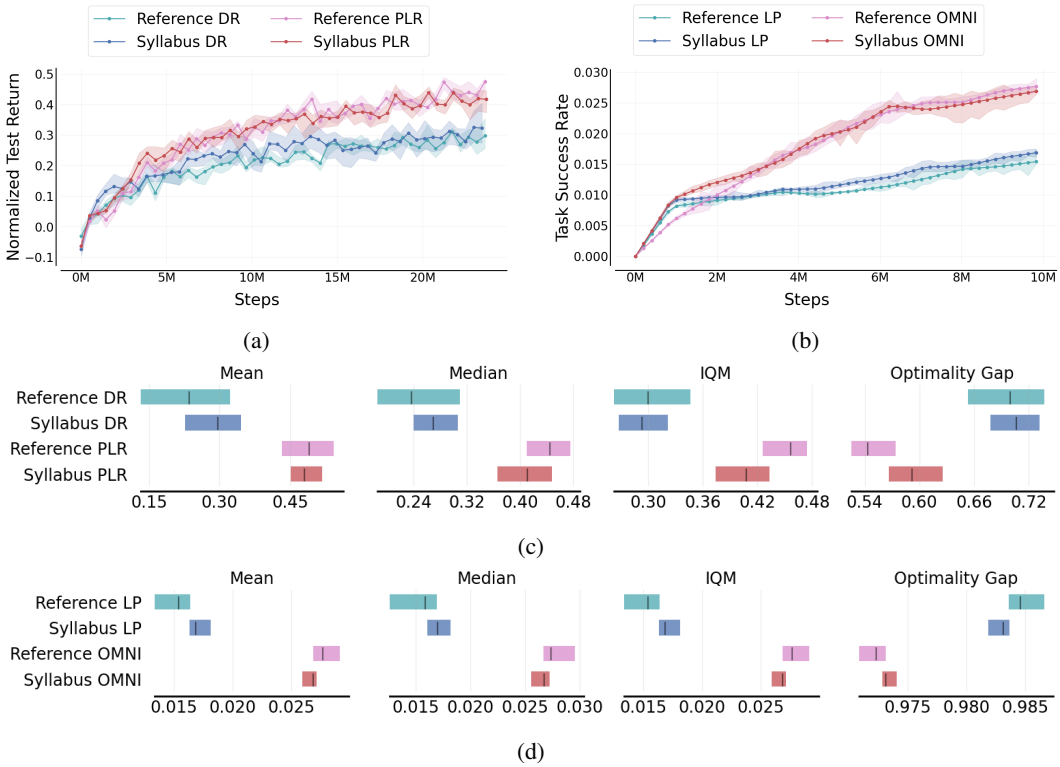


Figure 3: **(a)** Mean normalized test returns for Syllabus's implementation vs. the original implementation of Prioritized Level Replay from (Jiang et al., 2021b) on 10 Procgen environments. Domain Randomization is also included for reference. **(b)** Mean task success rate for Syllabus's Learning Progress and OMNI implementations vs. the reference implementations from (Zhang et al., 2023) on Crafter. **(c)** and **(d)** 95% Stratified bootstrap confidence intervals for Procgen and Crafter.

6 Reproduction Experiments

In order to demonstrate the correctness of Syllabus's infrastructure and algorithms, we reproduce the experiments for each automatic curriculum learning method from the paper in which they were introduced. We test PLR and DR on Procgen as in (Jiang et al., 2021b), and we test LP and OMNI on Crafter (Hafner, 2021) as in (Zhang et al., 2023). Syllabus makes it easy to test our implementations; we simply take any open-source reference codebase, disable the curriculum, and replace it with a few lines of Syllabus code. This lets us quickly evaluate new ACL implementations without recreating experiments in a new RL library, thereby minimizing methodological errors.

Procgen is a collection of 16 procedurally generated games designed to test sample efficiency and generalization (Cobbe et al., 2020a). Procgen levels are generated from a seed, so we use a task space of 200 training seeds for all curricula. Our experiments focus on a subset of 10 Procgen environments, but otherwise use the same methodology as Jiang et al. (2021b) and Cobbe et al. (2020a). Like these works, we normalize test returns by dividing returns by the empirical return range in each Procgen environment. Our full methodology is explained more thoroughly in Supplementary D.1. Crafter is a procedurally generated, grid-world environment modeled after Minecraft (Guss et al., 2019; Hafner, 2021). Agents collect resources, craft tools, and fight monsters to survive. The environment assigns tasks to the agent such as "collect 5 coal" or "make 1 iron pickaxe", which the agent receives 1 reward for completing. The curricula in Zhang et al. (2023) are therefore curricula over reward functions, based on the agent's current competence on each task.

We show in Figure 3 that the normalized test returns for PLR and DR on Procgen over the course of training almost precisely match the reference implementations from Jiang et al. (2021b). Similarly in Crafter, we see that our implementations of LP and OMNI achieve the exact same task success rates throughout training as the reference implementations. As in (Zhang et al., 2023) the task space includes 15 main tasks such as "collect 1 coal", 90 repeat tasks such as "collect 8 coal", and 1024 impossible tasks that always have 0 success rate. We evaluate the success rates for each of the 105 possible tasks for 4 episodes each over the course of training. These experiments provide strong evidence that our multiprocessing infrastructure and algorithm implementations are correct. The full details of our methodology for these experiments can be found in Supplementary D.1 and Supplementary D.2. We also use the Open RL Benchmark tools (Huang et al., 2024) to plot stratified bootstrap confidence intervals for the mean, median, and interquartile mean of each method as recommended by Agarwal et al. (2021).

Figure 3 shows that our implementations of PLR, LP, and OMNI match the performance of the reference implementations. Syllabus's portable design means that we can apply these algorithms to new domains with the confidence that the curriculum learning portion of the code is completely correct. In section 7 we demonstrate this by applying ACL to 2 new domains which have not previously been studied with curriculum learning.

7 New Baselines

We demonstrate Syllabus's versatility by applying ACL to two new complex domains, Neural MMO and NetHack, with baselines implemented in specialized RL libraries. We also present new baselines for each ACL method on Procgen and Crafter, trained using CleanRL (Huang et al., 2022) and TorchAC respectively. This is the first direct comparison using a shared benchmark of the learning progress methods LP and OMNI against UED methods like PLR and SFL.

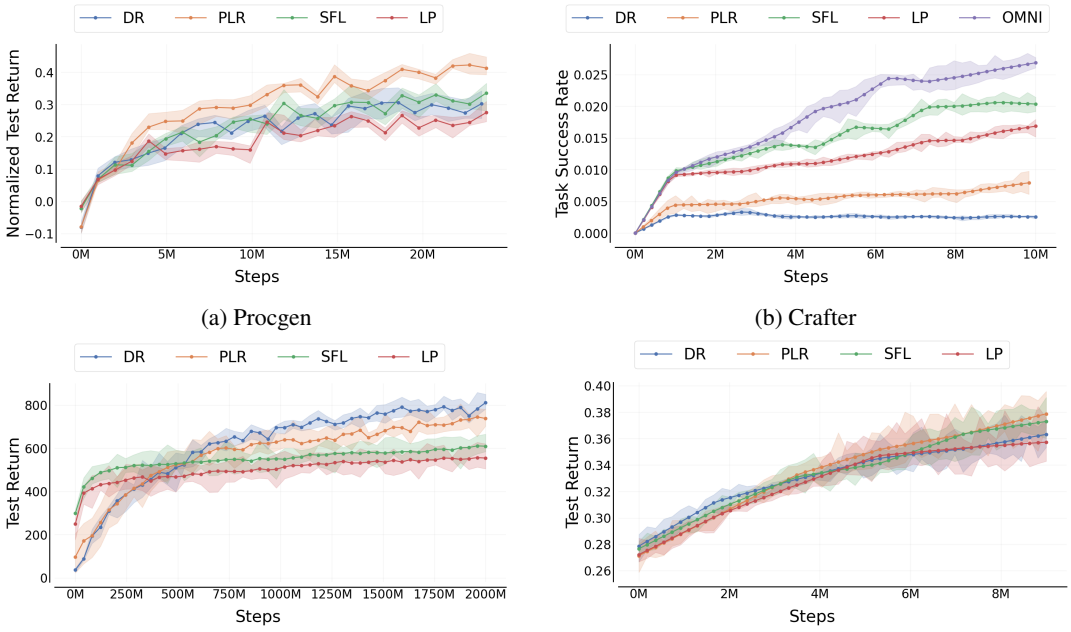
7.1 Neural MMO 2.0 in PufferLib

Neural MMO 2.0 is a complex multi-agent simulation inspired by massively multiplayer online games (Suarez et al., 2019; 2024). Agents can collect resources, learn skills, trade goods, and fight non-player characters or other agents. It has a predicate task space which allows users to define objectives in Python code. The baseline for the 2023 Neural MMO competition was written in PufferLib because it supports complex action spaces and multiagent environments where agents can die mid-episode, which complicates learning code (Terry et al., 2021; Suarez et al., 2024). We show that Syllabus can be used in this environment with 128 agents and a massive task space.

All of the automatic curriculum learning methods in Syllabus (excluding FSP and PFSP) were designed for single-agent environments. To apply single-agent curricula to a multi-agent environment, we update our curricula with feedback from all 128 agents controlled by our self-play policy. Each agent's experience is treated as trajectories from separate single-agent environments. We do not test FSP or PFSP because Neural MMO is a mixed competitive-cooperative game, not purely zero-sum.

7.2 NetHack in Moolib

NetHack is a popular text-based dungeon-crawler released in 1987, and adapted into an RL environment by Küttler et al. (2020). It's a complex, procedurally generated game in which winning or "ascending" can take more than 50,000 steps for human players. Ascending requires players to solve puzzles using common sense, knowledge of mythology, and game-specific tricks, all while collecting equipment, fighting monsters, and scavenging for food. NetHack remains one of the hardest benchmarks for online RL methods, which lag behind hand-crafted symbolic agents and behavior cloning baselines (Küttler et al., 2020; Tuyls et al., 2023; Piterbarg et al., 2024).



(a) Procgen

(b) Crafter

(c) NetHack

(d) Neural MMO

Figure 4: Automatic curriculum learning results on **(a)** Procgen, **(b)** Crafter, **(c)** NetHack, and **(d)** Neural MMO. 95% Stratified bootstrap confidence intervals can be found in Supplementary D

Importantly, NetHack can be simulated extremely quickly, shifting the training bottleneck from data collection to policy optimization and inference. To address this, NetHack baselines use specialized training libraries like TorchBeast (Küttler et al., 2019), Sample Factory (Petrenko et al., 2020), and moolib (Mella et al., 2022). Sample Factory and moolib use asynchronous PPO (APPO), which creates separate copies of the policy for action inference and optimization. Moolib spawns servers for policies and environments that communicate via remote procedure calls, allowing it to scale to many machines. We demonstrate that Syllabus is easy to use with moolib in a single GPU setting.

7.3 Methodology

LP and SFL rely on a task success metric to evaluate seeds, which is not a standard feature of RL environments. For Crafter, these are provided via binary pass/fail achievements. For all other environments, we define the success metric as the scaled, clipped, episodic return achieved by the agent. The exact scaling was manually selected based on the environment's typical reward range and is described in Supplementary D. OMNI is only evaluated on Crafter because it is not possible to identify "interesting" level seeds or maps by number alone, and there is no obvious way to describe arbitrary environment instantiations to an LLM. We use a grid search to tune every curriculum's hyperparameters in each environment, which we explain more thoroughly in Supplementary D.5.

Each of the libraries used in this paper have different design philosophies, software architectures, and multiprocessing implementations. Despite this, Syllabus can easily work with all of them with a few lines of library-agnostic code. We have an additional example in Supplementary E.1 of training CartPole agents in RLLib (Liang et al., 2018) to demonstrate how Syllabus can be used with Ray multiprocessing. We also experiment with adding PLR to Phasic Policy Gradients (Cobbe et al., 2021), an extension to PPO, in Supplementary E.3 to show that Syllabus is not limited to augmenting PPO. More details for all of our experiments can be found in Supplementary D, and the code and data are publicly available on GitHub and Weights & Biases respectively.

8 Results

We find that only PLR outperforms DR when applied to Procgen's level seeds. In this setting, LP underperforms and SFL, matching the simpler and cheaper DR. Neither of these newly evaluated methods perform well on Procgen despite hyperparameter tuning. We plot individual training curves for each Procgen environment in Figure 9 and find that there are no environments where LP or SFL outperform PLR, but there are some environments where they outperform DR. Also of note, DR is the only method which achieves nonzero return on Dodgeball. On our Crafter environment where 90% of tasks are impossible, the DR agent fails to learn a reasonable policy, barely outperforming and random policy. PLR is able to identify meaningful tasks and allows the agent to slowly improve its competence. Zhang et al. (2023) demonstrated the effectiveness of LP and OMNI on Crafter, but we find that SFL performs significantly better than LP. As a result, it seems natural to apply OMNI's LLM-based interestingness filtering on top of SFL instead of LP, but we found that this does not work well in practice Supplementary E.5. These results may suggest that task success rates are a more effective prioritization metric than value predictions when available. In their absence, value predictions might be a better approximation of competence than our return-based success metric defined in subsection 7.3. We leave a thorough investigation of this inconsistency as future work.

Crafter stands out as an effective testbed for CL research. Unlike other environments, it is easy to see which methods outperform others in Figure 4b. Crafter and Minecraft are designed such that proficiency on one task immediately makes the next task learnable, because the agent is guaranteed to have the knowledge and tools to acquire the next item. We suspect that this reward-based, linear task space highlights the strengths of CL algorithms. However, that nice structure is not easy to identify in most problems, so it is not a replacement for evaluating on complex environments.

In NetHack, we see that PLR performs similarly to DR. This is unsurprising because a single seed of NetHack can diverge significantly in just a few steps, so value predictions have extremely high variance and level seeds have little control over the environment instance's difficulty. However, we also see that LP and SFL accelerate learning at the start of training, but lead to lower asymptotic performance. As we saw in Crafter, success rates may be a more effective way of identifying learnable seeds, but prioritizing these seeds is not a good long-term strategy for boosting performance in NetHack. No curriculum provides any asymptotic benefits over DR in NetHack.

We also see in Figure 4d that none of our automatic curriculum learning methods for selecting maps in Neural MMO perform well compared to DR. In complex, many-agent environments the map may have minimal influence over the difficulty of the environment, suggesting that curricula over different dimensions could be more effective. Additional experiments using a manually designed curriculum of progressively harder reward functions can be found in Supplementary D, which we find outperforms the default survival reward on several metrics.

Overall, we find that most curricula perform poorly outside of the environment in which they were originally tested. Agents in both Procgen and Crafter have been shown to benefit from CL using PLR and LP respectively (Jiang et al., 2021b; Zhang et al., 2023), yet neither of these methods perform well when applied to the opposite environment. SFL is one of the few successful evaluations scoring higher than LP on Crafter and at least matching DR on Procgen, despite being developed on simpler JAX-based environments. None of the methods provide any noticeable improvement over the DR baseline in Neural MMO or NetHack. These results suggest that curricula over initial environment conditions are not effective in complex, long-horizon games, and that we may need to develop more sophisticated methods for these environments.

9 Discussion and Future Work

Multi-domain research, focusing on challenging, long-horizon environments, is crucial to developing generalizable and robust RL methods. Schaul et al. (2011), Bellemare et al. (2013), and Castro (2024) have argued that video game environments are an ideal benchmark for AI agents because they are designed for human players. This makes them inherently difficult yet learnable while be-

ing compelling and interpretable. In practice, solving these complex problems with reinforcement learning often requires custom infrastructure. NetHack agents are trained in libraries that can take advantage of its fast simulation speed. Neural MMO agents are built with PufferLib because it natively supports the complexities of multiagent interactions. Syllabus is designed for this reality of training agents in complex environments. Separating curriculum learning from RL code allows us to apply the same methods to new environments without re-implementation. We hope this will improve the reproducibility of curriculum learning research and help to push the field away from toy environments and toward domains that challenge modern RL methods.

Much of the recent UED infrastructure has been written in JAX. JAX allows environments and training code to be parallelized on hardware accelerators, producing experimental results hundreds of times faster than equivalent CPU-based environments. JAX is a powerful tool for conducting fast research, but it also enforces strict requirements on how code is written, slowing down development and incentivizing simplistic environments. For this reason, it will be challenging for JAX-based simulations to reach the complexity of CPU-based research environments, much less the professionally developed video games that have historically been benchmarks for reinforcement learning. While JAX has enabled rapid progress in curriculum learning using simplified environments, Syllabus is designed to extend these benefits to more complex domains.

Our results suggest that although curriculum learning is effective in moderately challenging single-agent environments like Procgen and Crafter, standard approaches may not be sufficient for complex, long-horizon, and multi-agent environments like NetHack and Neural MMO. These results could likely be improved by using curricula over more meaningful axes of the task space, such as reward functions in NetHack or opponent strategies in Neural MMO. However, more complex task spaces also introduce new challenges. For instance, training on multiple reward functions will require careful per-task reward normalization. One practical finding from our experiments is that SFL performs at least as well as LP in each environment, and only scores lower than DR in NetHack. In addition, our full distribution variant of SFL performs comparably to the original top K implementation and requires no hyperparameter tuning. Therefore, our full distribution implementation of SFL is a strong initial choice for exploring curriculum learning on new domains.

Syllabus is actively evolving as we add features, benchmark new methods, and expand support for general curriculum learning approaches. We hope it enables future research and encourages evaluation on more challenging domains. We hope it serves as a foundation for future curriculum learning research, particularly in challenging environments.

10 Limitations

Syllabus defines a completely separate multiprocessing pathway to send data to the curriculum. When curricula require observations, rewards, dones, or infos from the environment, this will send the same information as the RL training multiprocessing, potentially leading to bandwidth or processing bottlenecks. It is possible to avoid this duplication by writing additional code to forward data from the training process to the curriculum. Syllabus has not been tested on multi-node infrastructure, though it is possible to support multi-node systems via RPC calls with minimal modification. Syllabus's multiprocessing infrastucture cannot be used to communicate with environments vectorized in JAX or C without additional code, though the curricula can still be used directly. As a result, Syllabus can not be used to run the entire training system on hardware accelerators.

Syllabus does not currently implement any exploration bonuses, or advanced multiagent algorithms beyond self-play. Exploration bonuses typically train additional neural network to predict some measure of novelty (Bellemare et al., 2016a; Pathak et al., 2017b; Henaff et al., 2022) and use them to compute additional reward components. Multiagent methods using a protagonist and antagonist typically train another agent to propose tasks (Dennis et al., 2020; Sukhbaatar et al., 2018; OpenAI et al., 2021). Currently, Syllabus does not include methods that train additional networks or create new reward components. However, we expect that using those methods alongside Syllabus will be no harder than using them without Syllabus.

References

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 2021.

Christopher Bamford. Griddly: A platform for ai research in games. *Software Impacts*, 8:100066, 2021.

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. DOI: 10.1109/TSMC.1983.6313077.

Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016a.

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

Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pp. 1479–1487, Red Hook, NY, USA, 2016b. Curran Associates Inc. ISBN 9781510838819.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 41–48, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. DOI: 10.1145/1553374.1553380. URL https://doi.org/10.1145/1553374.1553380.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Michael Beukman, Samuel Coward, Michael Matthews, Mattie Fellows, Minqi Jiang, Michael Dennis, and Jakob Foerster. Refining minimax regret for unsupervised environment design. *arXiv preprint arXiv:2402.12284*, 2024.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

George W. Brown. Iterative solution of games by fictitious play. In T. C. Koopmans (ed.), *Activity Analysis of Production and Allocation*. Wiley, New York, 1951.

Pablo Samuel Castro. In defense of atari: The ale as a benchmark for autorl. 2024. URL https://icml.cc/virtual/2024/39301.

Seth Chaiklin. *The Zone of Proximal Development in Vygotsky's Analysis of Learning and Instruction*, pp. 39–64. Learning in Doing: Social, Cognitive and Computational Perspectives. Cambridge University Press, 2003.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

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

Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 2048–2056. PMLR, 13–18 Jul 2020b. URL https://proceedings.mlr.press/v119/cobbe20a.html.

Karl W Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. In *International Conference on Machine Learning*, pp. 2020–2027. PMLR, 2021.

Cédric Colas, Laetitia Teodorescu, Pierre-Yves Oudeyer, Xingdi Yuan, and Marc-Alexandre Côté. Augmenting autotelic agents with large language models. In *Conference on Lifelong Learning Agents*, pp. 205–226. PMLR, 2023.

Samuel Coward, Michael Beukman, and Jakob Foerster. Jaxued: A simple and useable ued library in jax. *arXiv preprint arXiv:2403.13091*, 2024.

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. *Advances in neural information processing systems*, 33:13049–13061, 2020.

Aaron Dharna, Charlie Summers, Rohin Dasari, Julian Togelius, and Amy K Hoover. Watts: Infrastructure for open-ended learning. In *ICLR Workshop on Agent Learning in Open-Endedness*, 2022.

Yuqing Du, Pieter Abbeel, and Aditya Grover. It takes four to tango: Multiagent self play for automatic curriculum generation. In *International Conference on Learning Representations*, 2023a.

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. In *International Conference on Machine Learning*, pp. 8657–8677. PMLR, 2023b.

Jeffrey L. Elman. Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99, 1993. ISSN 0010-0277. DOI: https://doi.org/10.1016/0010-0277(93)90058-4. URL https://www.sciencedirect.com/science/article/pii/0010027793900584.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. Omni-epic: Open-endedness via models of human notions of interestingness with environments programmed in code. *arXiv preprint arXiv:2405.15568*, 2024.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35: 18343–18362, 2022.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International conference on machine learning*, pp. 1515–1528. PMLR, 2018.

Alex Graves, Marc G Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *international conference on machine learning*, pp. 1311–1320. Pmlr, 2017.

William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*, 2019.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. Pmlr, 2018.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. *arXiv preprint arXiv:2109.06780*, 2021.

Eric Hambro, Sharada Mohanty, Dmitrii Babaev, Minwoo Byeon, Dipam Chakraborty, Edward Grefenstette, Minqi Jiang, Jo Daejin, Anssi Kanervisto, Jongmin Kim, et al. Insights from the neurips 2021 nethack challenge. In *NeurIPS 2021 Competitions and Demonstrations Track*, pp. 41–52. PMLR, 2022a.

Eric Hambro, Roberta Raileanu, Danielle Rothermel, Vegard Mella, Tim Rocktäschel, Heinrich Küttler, and Naila Murray. Dungeons and data: A large-scale nethack dataset. *Advances in Neural Information Processing Systems*, 35:24864–24878, 2022b.

Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pp. 805–813. JMLR.org, 2015a.

Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *International Conference on Machine Learning*, 2015b. URL https://api.semanticscholar.org/CorpusID:13937012.

Mikael Henaff, Roberta Raileanu, Minqi Jiang, and Tim Rocktäschel. Exploration via elliptical episodic bonuses. *Advances in Neural Information Processing Systems*, 35:37631–37646, 2022.

Shengyi Huang, Rousslan Fernand JulienDossa Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João GM Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *The Journal of Machine Learning Research*, 23(1):12585–12602, 2022.

Shengyi Huang, Quentin Gallouédec, Florian Felten, Antonin Raffin, Rousslan Fernand Julien Dossa, Yanxiao Zhao, Ryan Sullivan, Viktor Makoviychuk, Denys Makoviichuk, Mohamad H Danesh, et al. Open rl benchmark: Comprehensive tracked experiments for reinforcement learning. *arXiv preprint arXiv:2402.03046*, 2024.

Matthew T Jackson, Minqi Jiang, Jack Parker-Holder, Risto Vuorio, Chris Lu, Greg Farquhar, Shimon Whiteson, and Jakob Foerster. Discovering general reinforcement learning algorithms with adversarial environment design. *Advances in Neural Information Processing Systems*, 36, 2024.

Minqi Jiang, Michael Dennis, Jack Parker-Holder, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Replay-guided adversarial environment design. *Advances in Neural Information Processing Systems*, 34:1884–1897, 2021a.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, pp. 4940–4950. PMLR, 2021b.

Minqi Jiang, Ishita Mediratta, Mikayel Samvelyan, and Jayden Teoh. Dual curriculum design. https://github.com/facebookresearch/dcd, 2022.

Minqi Jiang, Michael Dennis, Edward Grefenstette, and Tim Rocktäschel. minimax: Efficient baselines for autocurricula in jax. *arXiv preprint arXiv:2311.12716*, 2023.

Ingmar Kanitscheider, Joost Huizinga, David Farhi, William Hebgen Guss, Brandon Houghton, Raul Sampedro, Peter Zhokhov, Bowen Baker, Adrien Ecoffet, Jie Tang, et al. Multi-task curriculum learning in a complex, visual, hard-exploration domain: Minecraft. *arXiv preprint arXiv:2106.14876*, 2021.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220 (4598):671–680, 1983. DOI: 10.1126/science.220.4598.671. URL https://www.science.org/doi/abs/10.1126/science.220.4598.671.

Pascal Klink, Haoyi Yang, Carlo D'Eramo, Jan Peters, and Joni Pajarinen. Curriculum reinforcement learning via constrained optimal transport. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 11341–11358. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/klink22a.html.

Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. Torchbeast: A pytorch platform for distributed rl. *arXiv preprint arXiv:1910.03552*, 2019.

Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684, 2020.

Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Joel Z Leibo, Edward Hughes, Marc Lanctot, and Thore Graepel. Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research. *arXiv preprint arXiv:1903.00742*, 2019.

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 *International conference on machine learning*, pp. 3053–3062. PMLR, 2018.

Ishita Mediratta, Minqi Jiang, Jack Parker-Holder, Michael Dennis, Eugene Vinitsky, and Tim Rocktäschel. Stabilizing unsupervised environment design with a learned adversary. *arXiv preprint arXiv:2308.10797*, 2023a.

Ishita Mediratta, Minqi Jiang, Jack Parker-Holder, Michael Dennis, Eugene Vinitsky, and Tim Rocktäschel. Stabilizing unsupervised environment design with a learned adversary. In Sarath Chandar, Razvan Pascanu, Hanie Sedghi, and Doina Precup (eds.), *Proceedings of The 2nd Conference on Lifelong Learning Agents*, volume 232 of *Proceedings of Machine Learning Research*, pp. 270–291. PMLR, 22–25 Aug 2023b. URL https://proceedings.mlr.press/v232/mediratta23a.html.

Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J Pal, and Liam Paull. Active domain randomization. In *Conference on Robot Learning*, pp. 1162–1176. PMLR, 2020.

Vegard Mella, Eric Hambro, Danielle Rothermel, and Heinrich Küttler. moolib: A Platform for Distributed RL. 2022. URL https://github.com/facebookresearch/moolib.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 561–577, 2018.

Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *The Journal of Machine Learning Research*, 21(1):7382–7431, 2020.

Alexander Nikulin, Vladislav Kurenkov, Ilya Zisman, Viacheslav Sinii, Artem Agarkov, and Sergey Kolesnikov. XLand-minigrid: Scalable meta-reinforcement learning environments in JAX. In *Intrinsically-Motivated and Open-Ended Learning Workshop, NeurIPS2023*, 2023. URL https://openreview.net/forum?id=xALDC4aHGz.

OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

OpenAI OpenAI, Matthias Plappert, Raul Sampedro, Tao Xu, Ilge Akkaya, Vineet Kosaraju, Peter Welinder, Ruben D'Sa, Arthur Petron, Henrique P d O Pinto, et al. Asymmetric self-play for automatic goal discovery in robotic manipulation. *arXiv preprint arXiv:2101.04882*, 2021.

Georg Ostrovski, Marc G. Bellemare, Aäron van den Oord, and Rémi Munos. Count-based exploration with neural density models. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 2721–2730. JMLR.org, 2017.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*, pp. 17473–17498. PMLR, 2022.

Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 2778–2787. JMLR.org, 2017a.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pp. 2778–2787. PMLR, 2017b.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning*, pp. 7652–7662. PMLR, 2020.

Ulyana Piterbarg, Lerrel Pinto, and Rob Fergus. Nethack is hard to hack. *Advances in Neural Information Processing Systems*, 36, 2024.

Rémy Portelas, Cédric Colas, Katja Hofmann, and Pierre-Yves Oudeyer. Teacher algorithms for curriculum learning of deep rl in continuously parameterized environments. In *Conference on Robot Learning*, pp. 835–853. PMLR, 2020a.

Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664*, 2020b.

Sebastien Racaniere, Andrew K Lampinen, Adam Santoro, David P Reichert, Vlad Firoiu, and Timothy P Lillicrap. Automated curricula through setter-solver interactions. *arXiv preprint arXiv:1909.12892*, 2019.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Carl Rasmussen. The infinite gaussian mixture model. In S. Solla, T. Leen, and K. Müller (eds.), *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/97d98119037c5b8a9663cb21fb8ebf47-Paper.pdf.

Clément Romac, Rémy Portelas, Katja Hofmann, and Pierre-Yves Oudeyer. Teachmyagent: a benchmark for automatic curriculum learning in deep rl. In *International Conference on Machine Learning*, pp. 9052–9063. PMLR, 2021.

Andries Rosseau, Raphael Avalos Martinez de Escobar, and Ann Nowe. Toward evolutionary autocurricula: Emergent sociality from inclusive rewards. In *From Cells to Societies: Collective Learning across Scales*, 2022. URL https://openreview.net/forum?id=BcUNSzkT-c.

Alexander Rutherford, Michael Beukman, Timon Willi, Bruno Lacerda, Nick Hawes, and Jakob Nicolaus Foerster. No regrets: Investigating and improving regret approximations for curriculum discovery. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration. *arXiv preprint arXiv:1812.03381*, 2018.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. DOI: 10.1147/rd.33.0210.

Mikayel Samvelyan, Akbir Khan, Michael D Dennis, Minqi Jiang, Jack Parker-Holder, Jakob Nicolaus Foerster, Roberta Raileanu, and Tim Rocktäschel. Maestro: Open-ended environment design for multi-agent reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2022.

Tom Schaul, Julian Togelius, and Jürgen Schmidhuber. Measuring intelligence through games. *arXiv preprint arXiv:1109.1314*, 2011.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/schulman15.html.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. DOI: 10.1038/nature16961.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Joseph Suarez. PufferLib: Making reinforcement learning libraries and environments play nice. In *Agent Learning in Open-Endedness Workshop at NeurIPS '23*, 2023.

Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural mmo: A massively multiagent game environment for training and evaluating intelligent agents. *arXiv preprint arXiv:1903.00784*, 2019.

Joseph Suarez, David Bloomin, Kyoung Whan Choe, Hao Xiang Li, Ryan Sullivan, Nishaanth Kanna, Daniel Scott, Rose Shuman, Herbie Bradley, Louis Castricato, et al. Neural mmo 2.0: A massively multi-task addition to massively multi-agent learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. In *6th International Conference on Learning Representations, ICLR 2018*, 2018.

Adrien Ali Taiga, William Fedus, Marlos C Machado, Aaron Courville, and Marc G Bellemare. On bonus-based exploration methods in the arcade learning environment. *arXiv preprint arXiv:2109.11052*, 2021.

Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.*, 10:1633–1685, December 2009. ISSN 1532-4435.

Adaptive Agent Team, Jakob Bauer, Kate Baumli, Satinder Baveja, Feryal Behbahani, Avishkar Bhoopchand, Nathalie Bradley-Schmieg, Michael Chang, Natalie Clay, Adrian Collister, et al. Human-timescale adaptation in an open-ended task space. *arXiv preprint arXiv:2301.07608*, 2023.

JK Terry, Benjamin J Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo De Lazcano Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, 2021.

Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995. ISSN 0001-0782. DOI: 10.1145/203330.203343. URL https://doi.org/10.1145/203330.203343.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 23–30. IEEE, 2017.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulao, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

Jens Tuyls, Dhruv Madeka, Kari Torkkola, Dean Foster, Karthik Narasimhan, and Sham Kakade. Scaling laws for imitation learning in nethack. *arXiv preprint arXiv:2307.09423*, 2023.

George Tzannetos, Bárbara Gomes Ribeiro, Parameswaran Kamalaruban, and Adish Singla. Proximal curriculum for reinforcement learning agents. *Transactions on Machine Learning Research*, 2023(5):1–21, 2023.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575:350 – 354, 2019. URL https://api.semanticscholar.org/CorpusID:204972004.

L. S. Vygotsky. *Mind in Society: Development of Higher Psychological Processes*. Harvard University Press, 1978. ISBN 9780674576285. URL http://www.jstor.org/stable/j.ctvjf9vz4.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

Rui Wang, Joel Lehman, Aditya Rawal, Jiale Zhi, Yulun Li, Jeff 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*, pp. 9940–9951, 2020.

Peter Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, Leilani Gilpin, Piyush Khandelwal, Varun Kompella, HaoChih Lin, Patrick MacAlpine, Declan Oller, Takuma Seno, Craig Sherstan, Michael Thomure, and Hiroaki Kitano. Outracing champion gran turismo drivers with deep reinforcement learning. *Nature*, 602:223–228, 02 2022. DOI: 10.1038/s41586-021-04357-7.

Mingqi Yuan, Roger Creus Castanyer, Bo Li, Xin Jin, Glen Berseth, and Wenjun Zeng. Rlexplore: Accelerating research in intrinsically-motivated reinforcement learning. *arXiv preprint arXiv:2405.19548*, 2024.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. Omni: Open-endedness via models of human notions of interestingness. *arXiv preprint arXiv:2306.01711*, 2023.

Ruize Zhang, Zelai Xu, Chengdong Ma, Chao Yu, Wei-Wei Tu, Shiyu Huang, Deheng Ye, Wenbo Ding, Yaodong Yang, and Yu Wang. A survey on self-play methods in reinforcement learning. *arXiv preprint arXiv:2408.01072*, 2024.

Zhuangdi Zhu, Kaixiang Lin, Anil K Jain, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

Supplementary Materials

The following content was not necessarily subject to peer review.

A API Reference

A.1 Curriculum API

```python
class Curriculum:
    """API for defining curricula to interface with Gym environments."""

    def _sample_distribution(self) -> List[float]:
        """ Returns a sample distribution over the task space.
            Any curriculum that maintains a true probability distribution
            should implement this method to retrieve the distribution. """

    def sample(self, k: int = 1) -> List[Any]:
        """ Sample k tasks from the curriculum. """

    def update_task_progress(self, task: Any, progress: Tuple[float, bool])
        """ Update the curriculum with a task and its progress.
            Progress is defined by the environment's TaskWrapper. """

    def update_on_step(self, obs: Any, rew: float, done: bool, info: dict):
        """ Update the curriculum with the environment outputs
            for the most recent step. """

    def update_on_episode(self, return: float, length: int, task: Any, env_id: int = None):
        """Update the curriculum with episode results from the environment."""

    def update_on_demand(self, metrics: Dict):
        """ Update the curriculum with arbitrary inputs.
            Typically used to incorporate gradient or error-based
            metrics from the training process. """

    def get_opponent(self, opponent_id: int) -> Agent:
        """ Load the agent corresponding to the given opponent_id. """

    def update_agent(self, opponent: Agent):
        """ Add opponent to agent store. """

    def update_winrate(self, opponent_id: int, opponent_return: int):
        """ Update the winrate for the given opponent_id based on environment returns. """
```

Figure 5: An abbreviated summary of the Curriculum interface. These represent the main methods for updating and sampling from a curriculum in Syllabus. The get_opponent, update_agent, and update_winrate methods are used for self-play.

A.2 Task Space API

```python
class TaskSpace:
    """ API for the range of tasks supported by an environment or
        curriculum learning algorithm."""

    def sample(self) -> Any:
        """ Sample a task randomly from the space. """

    def decode(self, encoding: Any) -> Any:
        """ Decode the task encoding to a format that can be interpreted by
            the environment. The task will be passed to the environment in this format."""

    def encode(self, task: Any) -> Any:
        """ Convert the task to an efficient encoding that can be interpreted by
            the curriculum. All curriculum updates receive the task in this format."""

    def seed(self, seed: int):
        """ Seed the task space for deterministic sampling. """

    @property
    def tasks(self) -> List[Any]:
        """ Returns a list of all tasks if task space is discrete """
```

Figure 6: Main features of the Task Space API.

A.3 Task Interface API

In unsupervised environment design, we study underspecified POMDPs (UPOMDPs), which have free configuration variables that need to be chosen to produce a fully specified POMDP (Dennis et al., 2020). In multi-task environments these free variables are the task, which can be a level seed, map specification, reward function, or even a new environment instance with different dynamics. Syllabus supports UPOMDPs by adding a `new_task` argument in the reset function of the standard Gym API. However, most environments do not support this behavior by default. We provide a TaskWrapper that accepts the new task, reconfigures the environment, then resets the environment for the next episode. This even allows us to add multi-task capabilities to single-task environments.

The Task Interface can also define an environment-specific progress metric to support curricula that depend on task success rates, and encode the current task into the observation space for task-conditional policies. As a result of this API structure, Syllabus is by far the easiest curriculum learning library to incorporate with preexisting training code, needing only a few lines of code to use complex ACL algorithms. Swapping between different curricula often requires only a single line of code change. Users can integrate new environments with a simple wrapper that tells Syllabus how to interpret its task space. We encourage the reader to look at our Procgen training script to better understand what Syllabus code looks like in practice. The next section explains the infrastructure that allows us to maintain this simplicity with asynchronous RL training infrastructure.

A.4 Sequential Curriculum

Syllabus implements a `Sequential` curriculum, which can be though of as a meta-curriculum over individual `Curriculum` objects. The curriculum has multiple stages, each of which will run until user-defined stopping conditions are met, after which the curriculum will move on to the next

```python
class TaskWrapper:
    """ Interface that changes the task assigned to a
        Gym or PettingZoo environment during reset."""

    def reset(self, *args, new_task: Any = None, **kwargs):
        """ Accepts a new task to be used in the next episode. """

    def change_task(self, new_task: Any):
        """ Modify the environment to use the new task. """

    def _task_completion(self, obs, rew, term, trunc, info) -> float:
        """ Implement this function to define a task progress metric. """

    def _encode_goal(self) -> Any:
        """ Encode the goal for the agent to observe. """

```

Figure 7: Main features of the Task Interface.

stage. The curriculum passes any updates that it receives to the current curriculum stage, so we can even use automatic curricula sequentially.

It is initialized with a set of stages, which can be individual elements of the task space, lists of tasks, entire task spaces, or other `Curriculum` objects. Individual tasks will be converted into a `Constant` curriculum which always returns the same task. Lists of tasks or task spaces are converted to `DomainRandomization` curricula over the provided tasks.

The stopping conditions can be predefined numbers of steps, tasks, or episodes, or they can be episodic return thresholds. We use a convenient text-based interface for defining stopping conditions which supports composite conditions. For example, the user can pass `"return>=1.0&&episodes>=1000"` as a valid composite conditions, which will stop the current stage once the agent has experiences 1000 episodes and achieves at least 1.0 return. The sequential curriculum receives updates from the environments to track agent progress automatically.

For an example of this sequential curriculum in practice, see Supplementary E.6.

B Optimization

As a consequence of the choice to use a separate multiprocessing system from the RL training loop, Syllabus incurs some unavoidable computational costs. Specifically, receiving and sending information in the environments decreases the effective steps per second of each environment, while sampling and sending tasks in the actor process increases the computational load on the main process. We evaluate Syllabus with NetHack to demonstrate the effect on overall steps per second. We use a minimal curriculum that always returns the same task to isolate the impact of our multiprocessing infrastructure.

The results are shown in Table 1 for 128 environments each running 64 episodes on a 2.20GHz 32-core Intel i9-13950HX [1]. We test Syllabus using both Python's native multiprocessing package and Ray (Moritz et al., 2018). Syllabus skips per-step updates for curricula that do not require them, instead only sending episodic data and tasks at the end of each episode.

[1] Syllabus is under continuous development, so these numbers may not reflect the performance of the most recent version of the library. They are accurate as of the time of publication.

Note that this is a worst case test for several reasons. Typically in asynchronous reinforcement learning, environments are vectorized and stepped together, such that N environments step at the speed of the single slowest environment. Here we run each environment independently, so they are not bottlenecked by vectorization. The NLE is an extremely fast environment with large observations, which stresses the multiprocessing communication bandwidth. Finally, RL is usually bottle-necked by policy optimization rather than environment iteration time. We expect Syllabus's impact on performance (as a percentage of total computation) to be much lower for more computationally intensive environments and when used in a real RL training context.

Table 1: Syllabus Performance Costs

Multiprocessing	Without Syllabus	Episodic Updates	Step Updates
Native Python	125s	131s (+4.8%)	150.0s (+20%)
Ray	135s	150s (+11.1%)	161s (+19.3%)

C Testing

We use pytest to continuously test and benchmark the performance of new additions to Syllabus on several environments. The multiprocessing infrastructure is evaluated to ensure that every sampled task is received by the environments and every environment updated is processed by the curricula, as well as several other safeguards. We use unit tests for task spaces and core curriculum features. We compare the performance of our algorithm implementations against the original implementations or original paper results whenever possible.

D Experiments

This section outlines the details of our experimental setup for each environment. All of the code for these experiments is also open-sourced on GitHub. Reinforcement learning research typically compares training returns to evaluate agents, but this is not valid when using curriculum learning. Curriculum learning modifies the training task distribution, meaning that higher returns may indicate that the curriculum prioritized easier tasks or tasks with larger return scales. In each experiment we separately evaluate agents on uniformly sampled tasks.

The experiments in this paper were run on consumer GPUs, mainly the GTX 1080Ti, RTX 2080Ti, and GTX Titan X GPUs. Each experiment was run on a single GPU. An estimate of the total cost of experiments for each environment is listed in Table 3. Note that this is only the cost to reproduce the exact plots in the main paper, not including any supplementary materials, hyperparameter tuning, or development time, which would make these estimates many times larger.

Table 2: Experiment Compute Resources

	Per Experiment		Total	
	CPU Hours	GPU Hours	CPU Hours	GPU Hours
Procgen	110	14	33000	4200
Crafter	224	14	6720	420
Neural MMO	352	22	7040	440
NetHack	960	60	19200	1200
LaserTag	192	12	2880	180

Table 3: Approximate cost to reproduce experiments in main paper.

D.1 Procgen

Our Procgen experiments use the same ResNet architecture and hyperparameters as previous work (Cobbe et al., 2020a; Jiang et al., 2021b). Procgen uses an action space of 15 discrete actions and produces e $64 \times 64 \times 3$ RGB observations. We use the exact same ResBlock model architecture, PPO hyperparameters, and PLR options as Cobbe et al. (2020a) and Jiang et al. (2021b) to reproduce their results with Syllabus's implementation of PLR. We experiment on a subset of 10 Procgen environments: Bigfish, Bossfight, Caveflyer, Chaser, Climber, Dodgeball, Fruitbot, Leaper, Ninja, and Plunder. We train 5 agents per environment on 200 seeds of the easy level distribution for 25M steps and evaluate them on the full distribution of seeds for 10 episodes every 16,384 environment steps. We compute normalized returns using the maximum and minimum return values for each environment listed in Cobbe et al. (2020a) according to the formula $r_N = \frac{r - r_{min}}{r_{max} - r_{min}}$. This allows us to weigh each environment equally while aggregating returns, such as in Figure 4a. For the LP, SFL, and OMNI curricula we compute the task reward as $min(max(r_N, 0.0), 1.0)$.

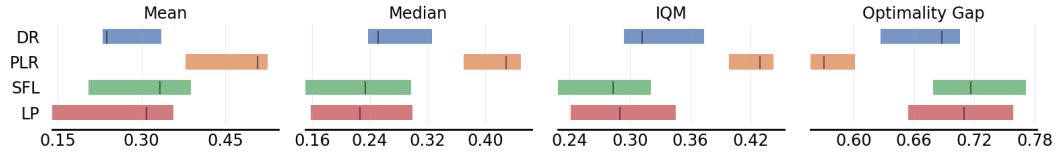


Figure 8: 95% Stratified Boostrapped Confidence Intervals for the Mean, Median, Interquartile Mean, and Optimality Gap of Normalized Test Returns of each ACL algorithm on Procgen.

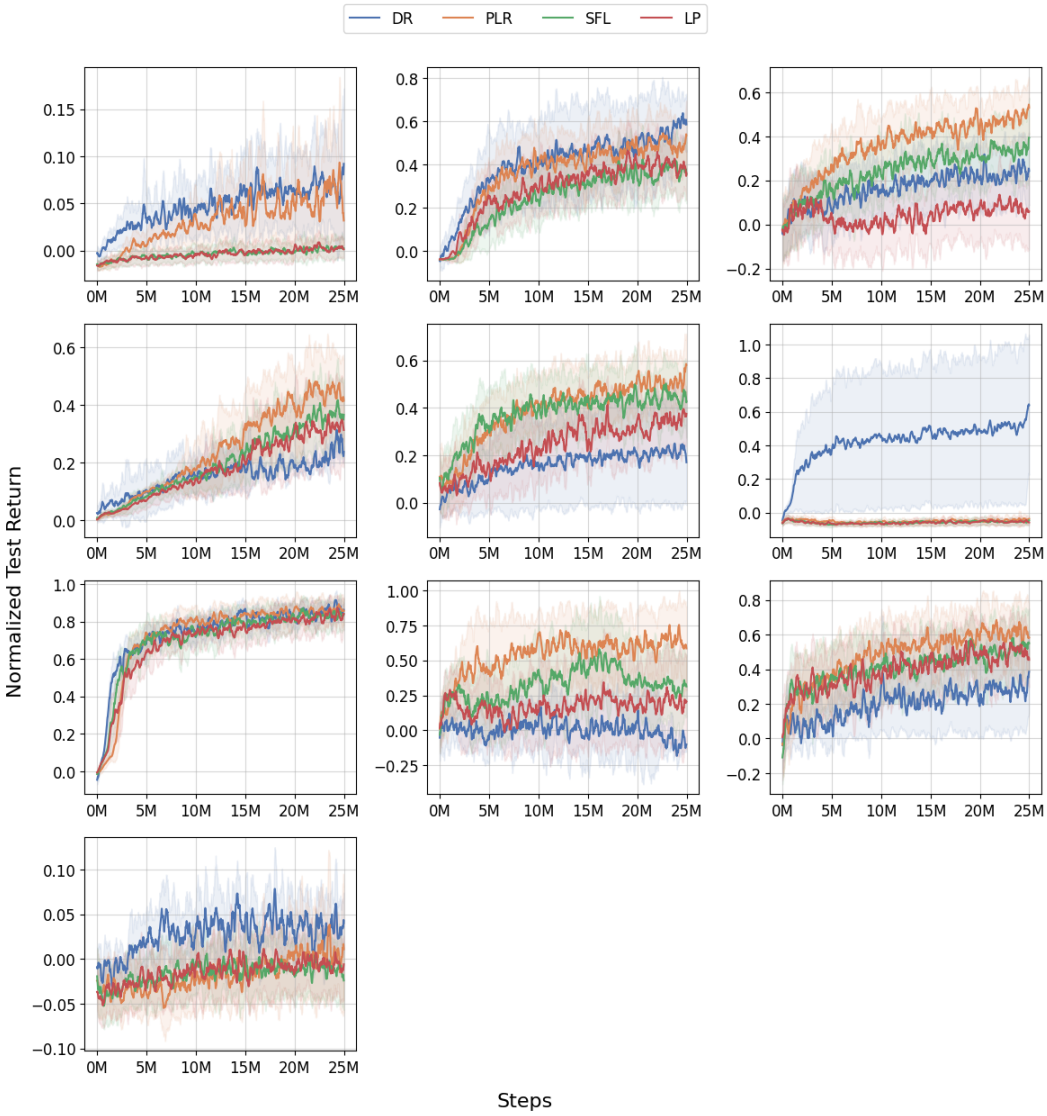


Figure 9: Normalized test returns for Domain Randomization, Prioritized Level Replay, Sampling for Learnability, and Learning Progress evaluated on 10 Procgen environments.

D.2 Crafter

For our Crafter experiments, we add Syllabus to the open-source training code from Zhang et al. (2023), meaning we use the same model architecture, hyperparameters, and Learning Progress arguments. Crafter has 13 discrete actions and 64 x 64 x 3 RGB observations. The observations are passed through a 2-layer CNN with ReLU activations followed by a fully connected layer of size 256. These visual embeddings are concatenated with a task encoding then passed into a 256 node LSTM layer. The actions and values are generated by separate 2-layer linear heads. We train agents for 10M steps and evaluate them every 25 updates (819,200 steps) on the full task space (excluding impossible tasks, which are assigned a success rate of 0).

As in Zhang et al. (2023) we use a task space of 15 unique tasks {"collect_wood", "collect_stone", "collect_coal", "collect_iron", "collect_diamond", "collect_drink", "make_iron_pickaxe", "make_iron_sword", "make_stone_pickaxe", "make_stone_sword", "make_wood_pickaxe", "make_wood_sword", "place_furnace", "place_stone", "place_table"}, 90 repeat tasks, including 10 repeat tasks for each "collect" task (e.g. "collect_9_coal"), and 5 repeat tasks each for "make" and "place" tasks, as well as 1024 impossible tasks. The impossible tasks have a fixed success rate of 0, and serve as distractions for the curriculum. This is a realistic representation of challenging tasks where most of the task space will be inaccessible to the agent until it learns advanced skills. Each ACL algorithm quickly learns to ignore these tasks while Domain Randomization samples them with the same frequency as the 105 real tasks. In this paper we report success rates including the impossible tasks, but the success rate over possible tasks can be recovered by multiplying our reported metrics by 10.75.

As in Kanitscheider et al. (2021) and Zhang et al. (2023) we train agent on the Simon Says task where the agent has 25 minutes (1500 steps) to complete as many tasks as possible in an episode. The agent has at most 5 minutes (300 steps) to complete any one task. If the agent succeeds, it is given a reward of 1 and assigned a new task. If it fails to complete the task in the time limit, it receives a reward of -1 and is assigned a new task. This Simon Says task in particular requires special consideration because we need to sample new tasks mid-episode, not just during the environment reset. Syllabus supports this behavior through it's synchronization wrappers.

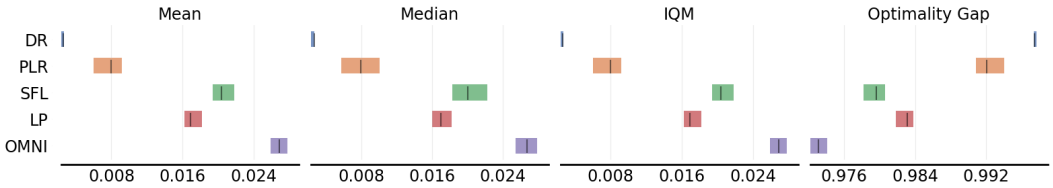


Figure 10: 95% Stratified Boostrapped Confidence Intervals for the Mean, Median, Interquartile Mean, and Optimality Gap of Normalized Test Returns of each ACL algorithm on Crafter.

D.3 Neural MMO

Our Neural MMO agents use the exact architecture and hyperparameters provided in the starter kit for the 2023 Neural MMO Competition Suarez et al. (2024). Neural MMO has complex dictionary action and observation spaces, so it uses separate CNN encoders for grid-based observations and fully connected encodings for vector observations, both with ReLU activations, as well as individual fully-connected heads for each action component. These agents are trained with self-play, with a single policy producing batched actions for 128 agents. For the LP, SFL, and OMNI curricula we compute the task reward as $min(max(max_a(R)/10.0, 0.0), 1.0)$, where a is an agent identifier and R is the mapping of agent identifiers to individual agents' mean episodic return because individual agents get close to but do not exceed 10.0 mean episodic return. We choose to prioritize tasks based on the most successful agent in the environment, but we did not explore other options. It is not clear what the best measure of a successful map is in Neural MMO and so we leave this investigation as future work.

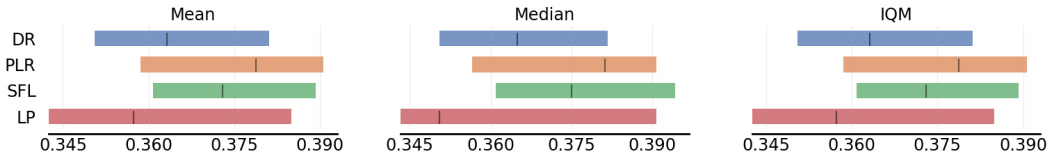


Figure 11: 95% Stratified Bootstrapped Confidence Intervals for the Mean, Median, Interquartile Mean, and Optimality Gap of Normalized Test Returns of each ACL algorithm on Neural MMO.

D.4 NetHack

Our NetHack agents are trained using the open source Moolib code from Hambro et al. (2022b). Moolib implements a version of Asynchronous Proximal Policy Optimization (APPO) (Schulman et al., 2017; Petrenko et al., 2020). We use the standard ChaoticDwarfGPT5 baseline from the NetHack Challenge (Hambro et al., 2022a), where the tty character observations are rendered to pixels and passed to the model in addition to text-based information. The model has a CNN with 4 layers and ELU (Clevert et al., 2015) activations for the image observation and linear encoders with ELU activations for the text components. We use the same training setttings as the NetHackChallenge environment, but we enable seeding to support curriculum learning. Notably, in this environment the episode terminates if the in-game timer does not progress for 150 agent steps, meaning the agent is taking meaningless actions or stuck interacting with menus. For the LP, SFL, and OMNI curricula we compute the task reward as $min(max(R/1000, 0.0), 1.0)$, where R is the mean episodic return because our agents get close to but do not exceed 1000 mean episodic return.

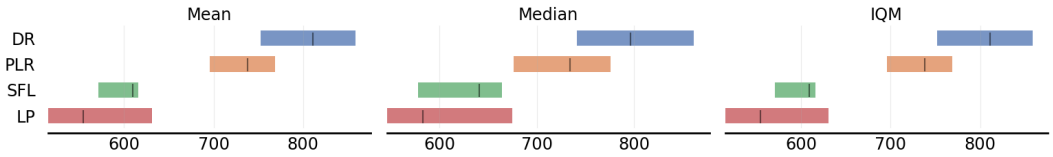


Figure 12: 95% Stratified Bootstrapped Confidence Intervals for the Mean, Median, Interquartile Mean, and Optimality Gap of Normalized Test Returns of each ACL algorithm on NetHack.

D.5 Hyperparameters

For each baseline in this paper, we use the tuned PPO hyperparameters from the work that introduced the code base. For each environment and curriculum learning method, we perform a separate grid search over the most important hyperparameters (as indicated by their respective papers). For PLR we search for the temperature parameter β in 0.1, 0.3, 0.5 and the staleness coefficient ρ in 0.1, 0.3, 0.5. For LP we search over 0.01, 0.1, 0.2, 0.3, 0.5 for the EMA $alpha$ parameter and over 0.01, 0.1, 0.3 for p_θ. For SFL we try sampling from the full distribution $p * (1 - p)$ and using the SFL method proposed in (Rutherford et al., 2024) which samples from the top K most learnable tasks with probability ρ and from the full task space with probability $1 - \rho$. We search for K in 10, 25, 50 and ρ in 0.5, 0.75, 1.0. Below is the table of all hyperparameters used in this paper.

Table 4: Learning Hyperparameters.

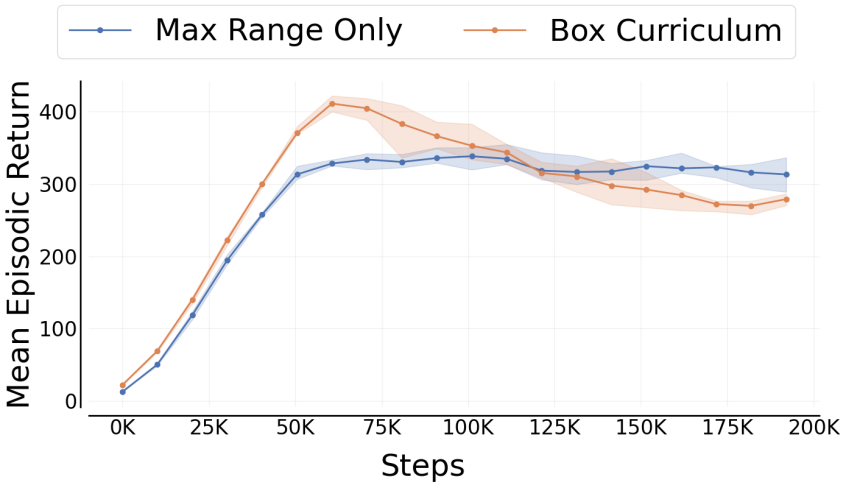
Parameter	Procgen	Crafter	NetHack	Neural MMO	LaserTag
PPO					
γ	0.999	0.99	0.999	0.99	0.995
λ_{GAE}	0.95	0.95	0.95	0.95	0.95
PPO rollout length	256	1024	32	128	256
PPO epochs	3	4	1	3	5
PPO mini-batches per epoch	8	16	1	32	4
PPO clip range	0.2	0.2	0.1	0.1	0.2
PPO number of workers	64	32	256	8	32
Adam learning rate	5e-4	1e-4	1e-4	1.5e-4	1e-4
Adam ϵ	1e-5	1e-5	1e-7	1e-6	1e-5
PPO max gradient norm	0.5	0.5	0.5	0.5	0.5
PPO value clipping	yes	yes	yes	yes	no
Return normalization	yes	no	yes	yes	yes
Value loss coefficient	0.5	0.5	0.5	0.5	0.5
Entropy coefficient	0.01	0.01	0.001	0.01	0.0
PLR					
Buffer size, K	200	200	200	128	-
Scoring function	VL1	VL1	VL1	VL1	-
Prioritization	rank	rank	rank	rank	-
Temperature, β	0.1	0.5	0.3	0.3	-
Staleness coefficient, ρ	0.1	0.1	0.1	0.3	-
LP					
EMA α	0.3	0.1	0.2	0.3	-
Reweighting p_θ	0.1	0.1	0.2	0.3	-
Update period, T	25	25	1000	50	-
SFL					
Batch size, N	200	200	200	128	-
Update period, T	25	25	1000	50	-
Sample method	Dist	Top K	Dist	Top K	-
Buffer size, K	-	10	-	25	-
Sample Ratio, ρ	-	1.0	-	0.5	-
FSP					
Agent checkpoint interval	-	-	-	-	800
PFSP					
f_{hard} entropy coef	-	-	-	-	2
smoothing constant	-	-	-	-	0.01
Win rate episodic memory	-	-	-	-	128

E Additional Experiments

E.1 RLLib with Syllabus

Cart Pole is a toy environment mainly used to debug RL implementations (Barto et al., 1983). It initializes a cart to a random starting point along a 2D track and tasks the agent with balancing a pole for as long as possible. It is not a multitask environment, so we use Syllabus's task wrapper to make the cart's initialization range a configurable option. This experiment demonstrates how Syllabus's Task Interface can add multitask functionality to singleton environments, and how Syllabus integrates easily with RLLib's Ray-based multiprocessing (Moritz et al., 2018; Liang et al., 2018).

Our Cart Pole experiments use a simple curriculum that increases the initialization range of the cart over the course of training. This causes the cart to begin in more precarious positions. We compare this to an agent trained only with the maximum initialization range. The curriculum learning agent initially learns a strong policy, but converges to a weaker policy than the agent trained solely on the maximum range, as we see in Figure 13a. Since the single-task agent easily converges to a strong policy, there is little reason to use curriculum learning.



(a) Training Cart Pole agents in RLLib, with curricula over the range of possible initial cart positions. We compare a simple curriculum of expanding the range throughout training vs. training with the fixed maximum initialization range.

E.2 Self-Play on LaserTag

We test our self-play algorithms (Self-Play, Fictitious Self-Play, and Prioritized Fictitious Self-Play) on the LaserTag environment introduced by Lanctot et al. (2017) and implemented by Samvelyan et al. (2022) in Griddly (Bamford, 2021). We use the same model architecture and hyperparameters as Samvelyan et al. (2022) and train our agents for 4000 updates or 65,536,000 environment steps, and add a copy of the current policy to the opponent buffer after every 800 updates. SP always plays against the current agent (agent 0), FSP uniformly samples from all past agents, and PFSP samples from all past agents based on the current agent's winrate against them. We see in Figure 14a that our methods properly update their sampling distributions after each agent is added to the opponent buffer. However, we do not see any difference between the methods against a fixed random agent in Figure 14b. We may need to train for longer to see any benefits to sampling historic agents.

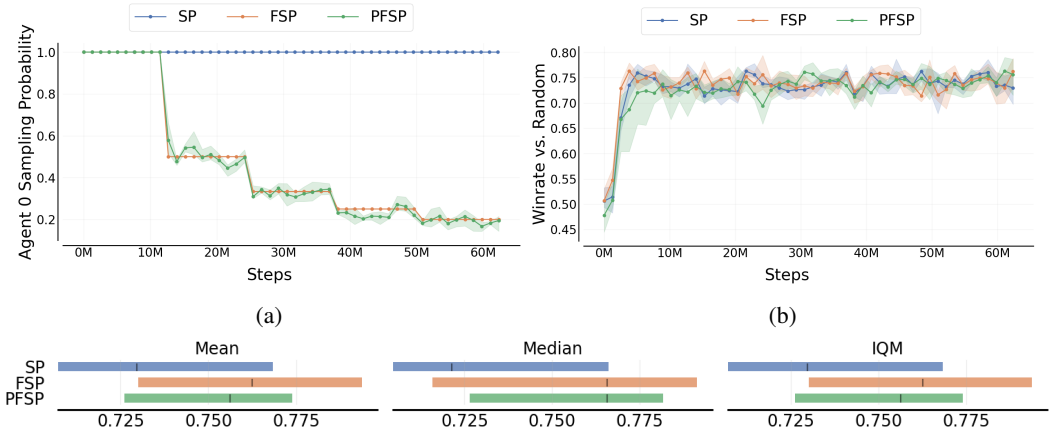


Figure 14: **(a)** Probability of sampling the first agent for each algorithm. Self-play only has one agent, so it always samples the current policy as it's opponent. **(b)** The winrate of each method against a fixed random policy. **(c)** Stratified bootstrapped confidence intervals of the winrates.

E.3 Phasic Policy Gradients

Phasic Policy Gradients (PPG) (Cobbe et al., 2021) is an extension of PPO that trains a separate value and policy network, while distilling features from the value network into the actor. They do this by adding an auxiliary value head to the policy, and periodically using behavior cloning from the value network into the auxiliary value head, allowing the actor to learn features used critic. Cobbe et al. (2021) showed that this approach outperforms PPO on Procgen. We use Syllabus study whether Prioritized Level Replay can provide an additional level of improvement over PPG as it does for PPO. PPG updates the policy and value with a separate number of epochs, but by default uses 1 value epoch and 1 policy epoch. We see in Figure 15 that applying PLR to PPG with the default hyperparameters performs worse than DR, despite its close similarity to PPO. We hypothesize that this is may be due to the lack of value updates. If the value predictions are less accurate, then PLR's score will also be inaccurate. We further investigate increasing the number of value epochs, and find that by increasing the number of value epochs to 3, the same as PPO, PLR matches but does not exceed DR.

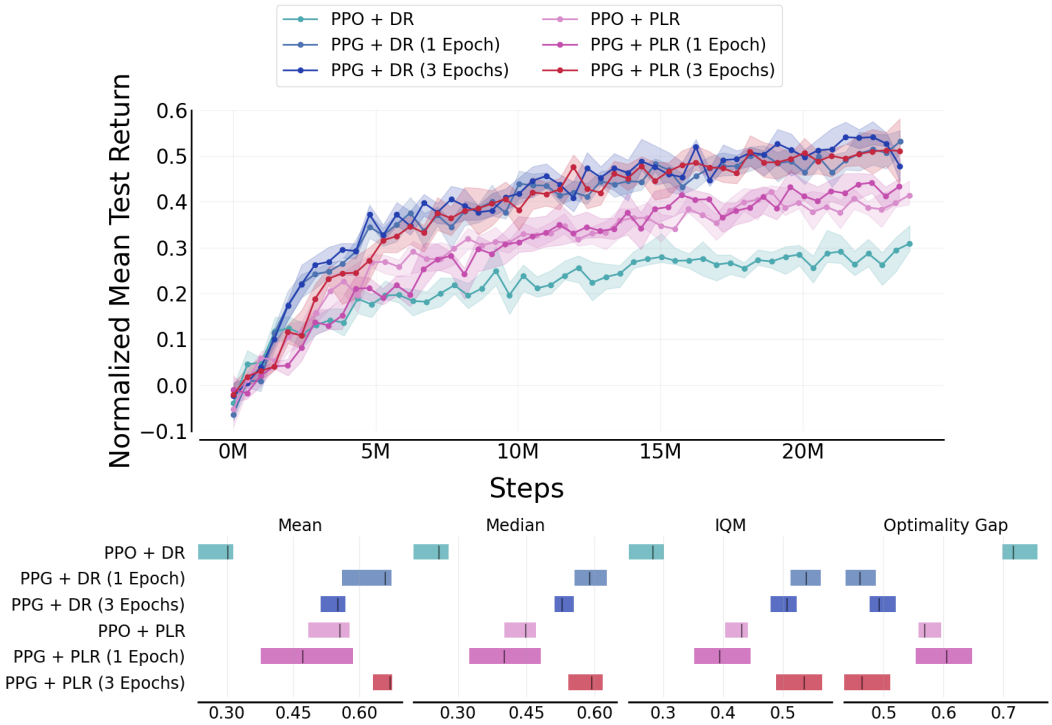


Figure 15: Normalized Test Returns of PPO and PPG with 1 or 3 value epochs when trained with DR or PLR on 10 Procgen environments with 5 seeds each.

E.4 Stale Value Predictions

We continue the investigation from Supplementary E.3 and further explore the importance of value prediction quality by training on delayed tasks with PLR. This is equivalent to sampling from a distribution calculated from stale value predictions. We can use Syllabus's asynchronous sampling code to artificially increase the delay. We find in Figure 16 that as the delay increases, the performance of PLR drops until it is nearly equivalent to domain randomization.

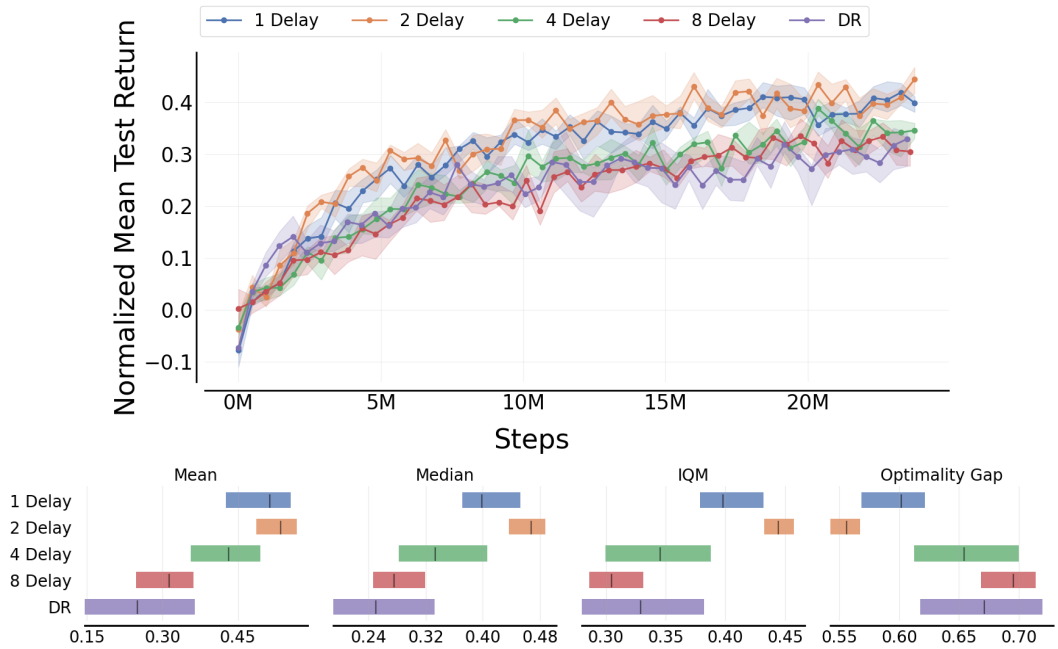


Figure 16: Training Procgen agents with PLR using stale value predictions. 1 buffer is equivalent to 1 episode of delay per environment.

E.5 Sampling for Learnability with OMNI

Sampling for Learnability and Learning Progress are very similar methods in terms of implementation. Both periodically evaluate the agent over the entire task space to generate task success rates, then generate a sampling distribution from those rates. OMNI adds an additional component to the LP curriculum by using an LLM to filter non-interesting tasks out of the distribution. In Figure 4b we found that SFL outperforms Learnability on Crafter, so we chose to investigate whether OMNI's filtering step could also improve the performance of SFL. Unfortunately we find that neither the full distribution nor top K implementations of SFL benefit from OMNI's filtering, as seen in Figure 17.

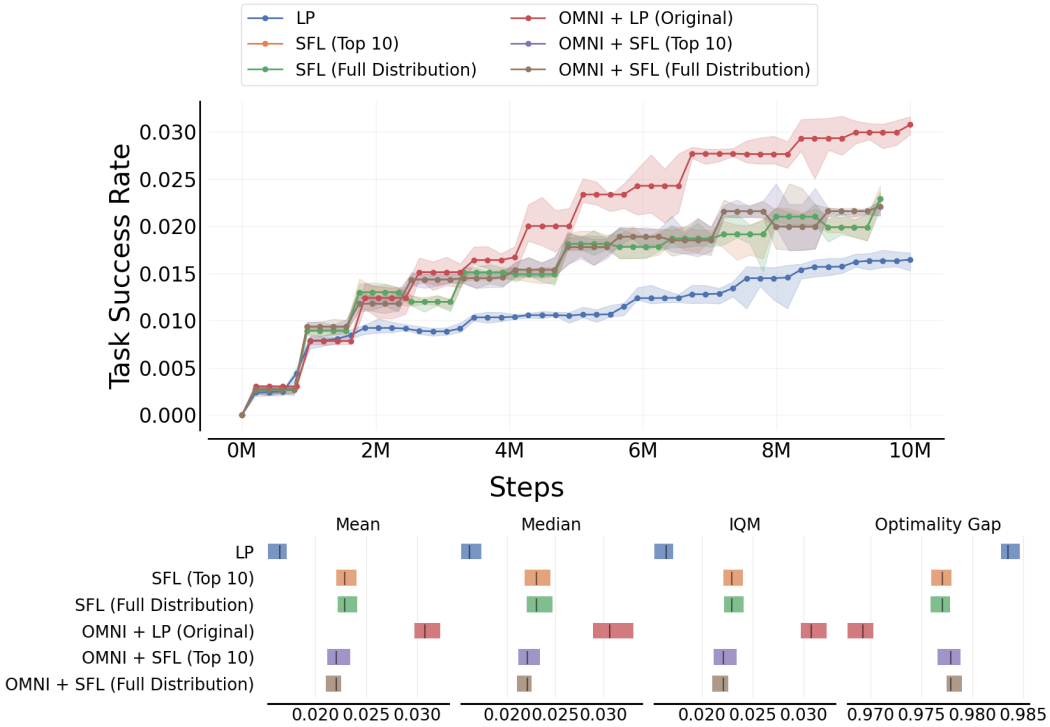


Figure 17: Mean task success rates for the Full Distribution and Top K implementations of SFL with OMNI's interestingess filter. LP and OMNI are shown for reference.

E.6 Neural MMO Task-Based Curriculum

We rely on events and achievements that are not zero-sum to determine how proficient each agent is in the environment, but these metrics are not completely independent from the quality of opponents. For example, the frequency of dying to starvation may decrease because agents are becoming better at scavenging for food, or because they are becoming more proficient at fighting, increasing their chance of dying to combat rather than starvation.

The sequential curriculum consists of 5 stages, each of which uses domain randomization over a subset of the task space. The curriculum progresses to the next stage whenever the agent achieves a mean episodic return of 0.75 averaged over the past 1000 episodes. Each agent successfully made it to the final curriculum stage by the end of training.

For each individual task, we assign a threshold. The agent is gets a reward of 1.0 for completing the task, which is distributed as the agent makes progress on the task. For example, if we task the agent with surviving for 100 timesteps, after 50 timesteps it will have a cumulative reward of 0.5. After surviving for 150 timesteps, the agent will have a cumulative reward of 1.0 because we stop assigning reward after the threshodl is reached. The tasks and thresholds for each stage are listed in Table 5

Table 5: Neural MMO Sequential Curriculum Task Thresholds

Task	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
Survive (steps)	50	150	250	350	500
Eat Food (count)	5	15	25	35	50
Drink Water (count)	5	15	25	35	50
Harvest Item (count)	3	9	15	21	30
Go Far (distance)	3	9	15	21	30
Level Up (count)	2	6	10	14	20
Equip Item (count)	1	3	5	7	10
Consume Item (count)	1	3	5	7	10
Buy Item (count)	1	3	5	7	10
Player Kill (count)	1	3	5	7	10

We see in Figure 18 that the sequential curriculum seems to create more aggressive before as these agents harvest and equip more weapons and ammo, and die to attacks more frequently. On the other hand, agents trained with domain randomization utilize the weapon and item system at similar rates to the other agents, but fail to learn some game systems like earning gold and killing NPCs.

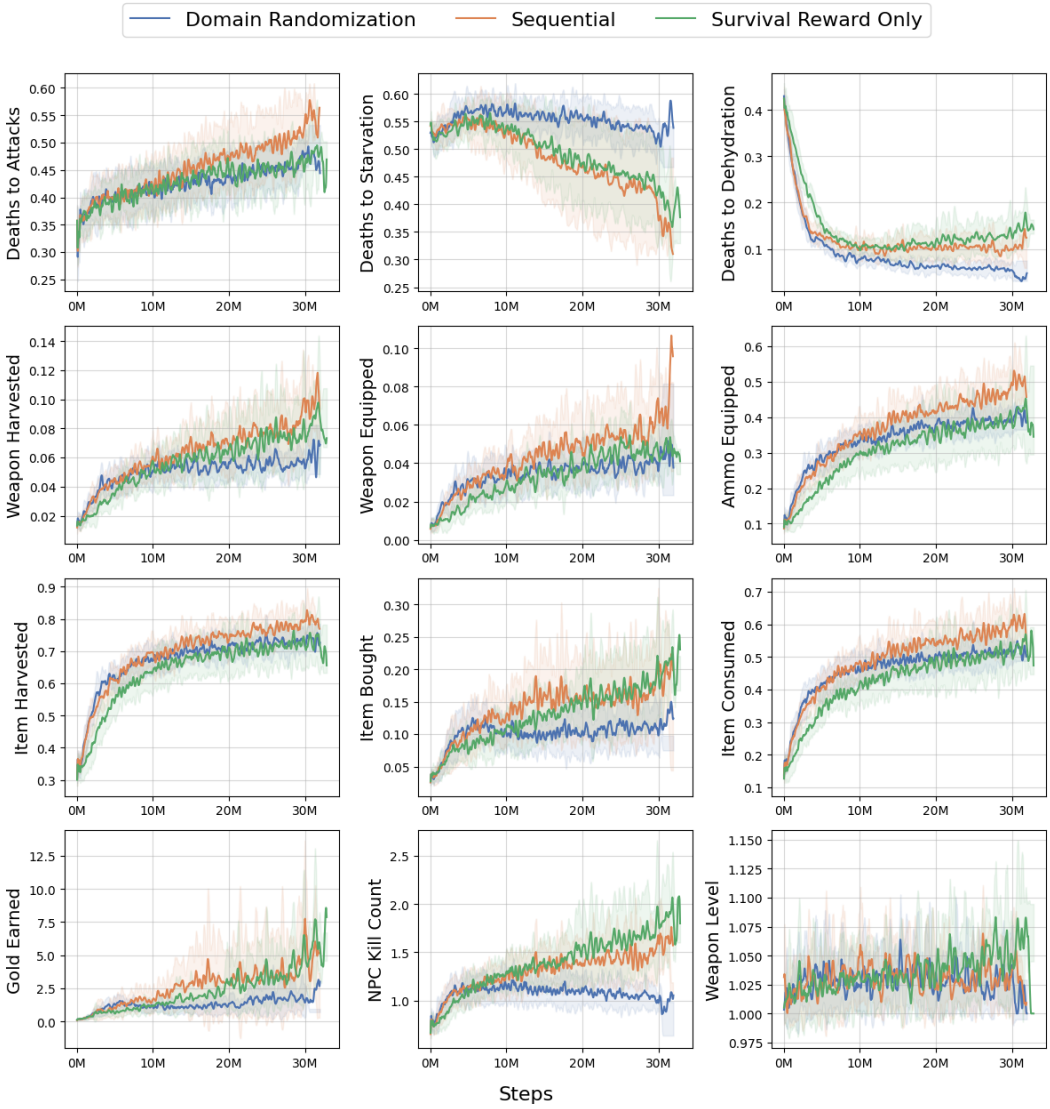


Figure 18: Various events and achievements in Neural MMO for Domain randomization and a manually designed sequential curriculum over the course of training.

F Code Examples

F.1 RLLib

Figure 19: Adding curriculum learning with Syllabus to RLLib training code with just a few lines of code.

```
    import gym
    from ray.tune.registry import register_env
    from ray import tune
    from gym.spaces import Box
    from .task_wrappers import CartPoleTaskWrapper

+   from syllabus.core import RaySyncWrapper, make_ray_curriculum
+   from syllabus.curricula import SimpleBoxCurriculum
+   from syllabus.task_space import BoxTaskSpace

    if __name__ == "__main__":
+       # Define a task space
+       task_space = BoxTaskSpace(Box(-0.3, 0.3, shape=(2,)))

        def env_creator(config):
            env = gym.make("CartPole-v1")
+           # Wrap the environment to change tasks on reset()
+           env = CartPoleTaskWrapper(env)
+           # Add environment sync wrapper
+           env = RaySyncWrapper(env, task_space=task_space)
            return env

        register_env("task_cartpole", env_creator)

+       # Create the curriculum
+       curriculum = SimpleBoxCurriculum(task_space)
+       # Add the curriculum sync wrapper
+       curriculum = make_ray_curriculum(curriculum)

        config = {
            "env": "task_cartpole",
            "num_gpus": 1,
            "num_workers": 8,
            "framework": "torch",
        }

        tuner = tune.Tuner("APEX", param_space=config)
        results = tuner.fit()

```

F.2 Stable Baselines 3

```
    import gym
    import procgen  # noqa: F401
    from stable_baselines3 import PPO
+   from syllabus.core import make_multiprocessing_curriculum
+   from syllabus.curricula import DomainRandomization
+   from syllabus.examples.task_wrappers import ProcgenWrapper
+   from syllabus.task_space import TaskSpace

    if __name__ = "__main__":
        def make_env(curriculum, task_space):
            def thunk():
                env = gym.make("procgen-bigfish-v0")
+               # Wrap the environment to change tasks on reset()
+               env = ProcgenWrapper(env)
+               # Add environment sync wrapper
+               env = MultiProcessingSyncWrapper(
+                   env,
+                   curriculum.components,
+                   task_space=task_space,
+               )
                return env
            return thunk

+       # Define a task space
+       task_space = DiscreteTaskSpace(200)
+       # Create the curriculum
+       curriculum = DomainRandomization(task_space)
+       curriculum = make_multiprocessing_curriculum(curriculum)

        venv = DummyVecEnv(
            [
                make_env(curriculum, task_space)
                for i in range(64)
            ]
        )

        model = PPO("CnnPolicy", venv)
        model.learn(25000000)

```

G Documentation

Syllabus is documented both in code and with a dedicated documentation website. This includes details on each curriculum algorithm and warnings about common pitfalls that users might run into when configuring them. A sample of the documentation website is shown in Figure 20.

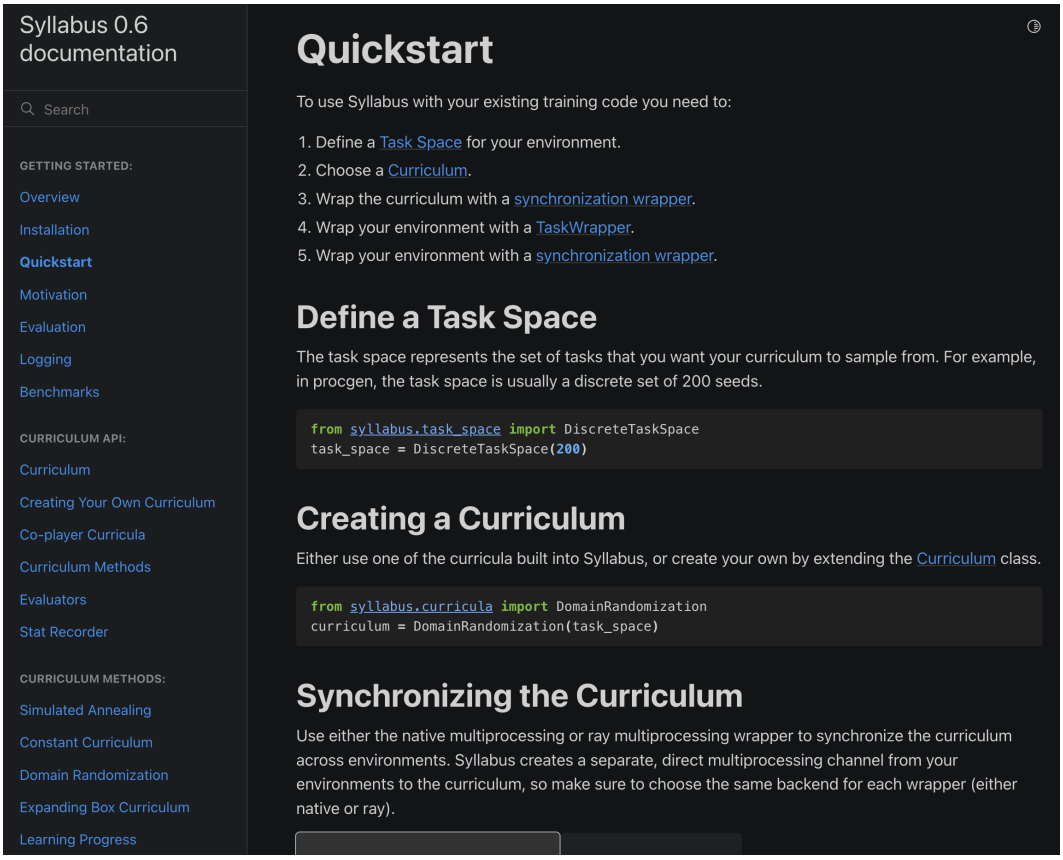


Figure 20: Quickstart page of Syllabus's documentation website.