A Study Into Domain Generalization for Reinforcement Learning

Jacob Le Mons (Jake Lemons)

5/10/23

Abstract

We evaluate and compare our own agents trained with low-intensity training to high-intensity trained agents from the Procgen Benchmark [1], [2]. Using the same fundamental tools as were used in the Procgen Benchmark including its framework and the Proximal Policy Optimization (PPO) algorithm [3], we explore my personal investigation into Reinforcement Learning (RL), emerging abilities of RL agents to generalize to new environments, and how low-intensity training compares to industry level high-intensity training for successful performance. We additionally used a personal GitHub repository [4] for saving and sharing our work as well as StableBaselines3 [5], [6] for training metric logging [7] and algorithm implementation tools [8], [9]. Low-intensity training had significantly better results than anticipated, but was also unreliable in creating well-performing agents consistently. Based on our results however, it's clear that independent of initial training performance, agents that are given enough training time will eventually succeed consistently. The amount of time that it takes for successful performance increases as the number of different environments the agents train on also increases. We observed little to no emerging generalization techniques within our agent sets, but were pleasantly surprised by some of our agents' performance given their minuscule level of training when compared to those described in the Procgen Benchmark. While not a traditional research paper, this paper was written with the intent of striking a balance between technical analysis and comprehensive overviews of various fundamental Artificial Intelligence (AI) topics, valuable for anyone operating in our rapidly changing modern world.

1. Introduction

How can Artificial Intelligence (AI) develop generalizable skills through reinforcement learning (RL)? In the last decade, lots of work and innovation has been done by AI researchers and engineers to explore ways of answering this question [2]. And for good reason too. Just think about how exciting it is to watch a Boston Dynamics robot do something that felt unrealistic just five or ten years ago. The world undergoes constant change and feels like it's getting more and more complicated every day. How then can we expect the AI models we create to contend with an increasingly complex world that even the people designing these models struggle to keep up with?

Drawing inspiration from the generalization question, this paper is my personal investigation into exploring a similar question: *how well can RL models generalize to new environments with low-intensity training?* We analyze the effects of low-intensity training of RL agents using OpenAI's Procgen Benchmark framework [1] and their Proximal Policy Optimization (PPO) [3] algorithm with implementation and analysis tools from StableBaselines3 [6]. This paper explores how the results of low-intensity training collected for this paper compare to those presented in the Procgen Benchmark. Do certain training variables have a larger effect on performance than others? What are some potential ways to increase the effectiveness of low-intensity training? Let's contemplate these questions and more throughout this paper.

After deciding to do a project on RL, we used OpenAI's Gymnasium [10] as a starting point. We played around with their provided tools and then looked into OpenAI's Procgen Benchmark [1], which we eventually settled on working with. Using the OpenAI Procgen Benchmark Article [1] and Paper [2], we developed a basic understanding of Procgen. We constructed a collaborative work environment using a virtual environment [4], Visual Studio Code [4], and GitHub [4] to share and save our work. With the help of OpenAI's Procgen GitHub [11], we implemented the Procgen CoinRun game environment and played it interactively to gain a better understanding of the game environment before testing out the RL with a basic random-action agent. We then implemented the PPO algorithm [9], and began training agents. Two sets of agents were trained: the initial set with 18 agents and the experimental set with 40 agents. We used StableBaselines3's Tensorboard integration [7] for logging training metric data and visually observed the testing performance as trained agents attempted levels in the rendered CoinRun game [11]. Using the metric logs, visualized gameplay performance, and a comparison of results between agents, agent sets, and the Procgen Benchmark, we evaluated the agents.

2. Background

General Overview of RL

RL is a method of machine learning (ML) where an AI model is put into an environment and provided with actions that it can make to change the state of the environment. These—often virtual—environments generally include games or are game-like simulations. The **agent** is given a **policy** (a function that maps from state to action, essentially describing how the agent acts). The simplest policies are the do-nothing policy (agent takes no action), a random-action policy (agent takes random action), and a greedy policy (agent tries to maximize reward without regard for future past one action). It's no surprise that these yield little to no chance of producing a well-performing agent, so more complex and robust policies are used instead. Some policies focus more on incentivizing things like exploration, while others give the agent a bit more freedom and don't restrict its learning quite as much. To explore some of the most influential RL game models of the past decade, see AlphaStar [12] and KataGo [13].

Procgen Benchmark From OpenAI

Procgen is short for procedural generation which is a computational rendering technique often used in video game development. The Procgen Benchmark is OpenAI's combination of 16 retro-style game environments, each with different gameplay mechanics and end goals. Additionally, it includes OpenAI's analytics and results of high-intensity training and testing in all 16 games using procedural generation for random level development. OpenAI's goal with the Procgen Benchmark was to develop further understanding of how procedural generation and RL incentivises agents to develop generalizable skill sets to succeed in a vast variety of game

environment levels. CoinRun is one of the 16 game environments and is the only one used in this project. In OpenAI's Procgen Benchmark Article section *Evaluating Generalization*, they write, "We came to appreciate how hard RL generalization can be while conducting the Retro Contest, as agents continually failed to generalize from the limited data in the training set. Later, our CoinRun experiments painted an even clearer picture of our agents' struggle to generalize." They further discuss the impact training set size has on generalization. For all 16 game environments, they used training sets ranging from 100 to 100,000 levels and "trained agents for 200M timesteps on these levels using Proximal Policy Optimization" before measuring performance on unseen test levels. Next, they found that for almost all game environments, "agents strongly overfit to small training sets". In order to close the generalization gap, "in some cases, agents need access to as many as 10,000 levels". Refer to Figure 1 to see a graph of the generalization on CoinRun. Their following point illustrates that they found an exciting trend emerge, "past a certain threshold, training performance improves as the training sets grows [sic]!" They initially observed this trend with the CoinRun environment, but pointed out that "it often happens in other Procgen environments as well." Finally, within this article, OpenAI also notes that they optimized environments to perform thousands of timesteps per second, something we didn't do for our low-intensity training.

Figure 1: Graph from Open AI's Procgen Benchmark showing the generalization performance of agents in CoinRun. The x-axis represents the number of levels and the y-axis represents the mean reward. Notice that the training (blue, starts higher on y-axis) and testing (orange, starts lower on y-axis) curves converge right around 10,000 levels.

PPO Algorithm

PPO is a collection of policy gradient methods for RL. The PPO algorithm is the combination of these methods. Quickly after its release, PPO became one of the most popular and widely applicable RL algorithms due to the fact that "PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time." [14] (Note that "wall-time" can be summarized as the amount of real-world time it takes for a computer to complete its assigned task.) Policy gradient methods have been vitally influential in AI breakthroughs over the past decade, but also have their fair share of challenges. As OpenAI states in their PPO article, "...getting good results via policy gradient methods is challenging because they are sensitive to the choice of stepsize" [3]. Progress is too slow if the stepsize is too small, and major performance decreases or signal noise becomes a challenge when the stepsize is too large. Their sample efficiency, that is, the amount of timesteps it takes to learn a task, is also prohibitive. Other algorithms such as Trust Region Policy Optimization (TRPO) and Actor Critic with Experience Replay (ACER) were created to solve some of these problems. Cornell University does a great job explaining the application of a trust region method as, "a powerful method that can update the objective function in each step to ensure the model is always getting improved while keeping the previously learned knowledge as the baseline" [15]. An actor-critic method combines the use of an agent (the actor) and an evaluator (the critic) where the actor proposes a range of possible actions based on its environment state and then the critic evaluates those actions based on its policy [16]. The Advantage Actor Critic (A2C) algorithm is another example of an actor-critic method and was also used for the development of PPO. TRPO is the predecessor to PPO. TRPO is only especially useful in specific tasks such as control tasks, but can be difficult to implement with certain other algorithms, while ACER is just significantly more complicated than PPO. In addition to its other attributes, PPO combines the most effective parts of these and other algorithms, making it easier to use, more widely applicable, and yielding similar if not better performance than other leading algorithms.

Simulation Gap

One of the biggest challenges in RL is what's known as the simulation gap. It can be explained as the gap between RL in a simulated environment versus RL in a physical real-world environment. Many of the companies that use RL to train robots have incredibly robust and intricate simulated environments that they use for training. Training in a real-world environment usually takes significantly longer to learn because of the added factors these companies must account for. Examples of additional factors may include robot hardware maintenance and the inability to simulate/train multiple iterations of a model simultaneously, unless multiple physical versions of the robot exist (which adds additional resource management difficulties and more complexity overall). While these robust simulators can't perfectly simulate real-world environments, they're used because they get fairly close to being able to do so. The simulators are usually adequate as a training starting point for the physical tasks RL is currently tackling such as walking robots. In a simulation, the primary advantage is the ability to speed up time

because you're really only limited by computing power. Now, there isn't a "best way" to do RL. Even within the ML sub-discipline RL, there are so many different methods and approaches for development. It all comes down to what you're trying to accomplish. For games such as Chess, Go, or Starcraft II, a physical real-world environment would probably be considered a waste of resources and unnecessary. For something like self-flying drones on the other hand, the physical real-world training approach may be seen as being a better approach than exclusively a simulated environment because the end goal is to have the drones flying in the real world on their own. However, it's just overall more difficult to accurately simulate robots interacting with their environment than it is to simulate a board game. It's still a major challenge to get a physically accurate simulation of objects that make and break contact with their environment, such as a foot of a walking robot. For the drone example, it's even more challenging to simulate complexities like the turbulent aerodynamics involved with high-speed drone flight. It's areas like this where the underlying physics, of turbulent air in this case, are still in the process of being understood. So while physical testing and complex physical simulation both have their own caveats, an argument can be made in support of using both methods of training together utilizing the advantages of each method. When broken down, we can see that exploring the effects of these different approaches on their own and in combination is the best way to optimize the potential for developmental success down the line.

Markov Decision Process

The Markov Decision Process (MDP) is the foundation for reinforcement learning and sequential decision making. The MDP offers a mathematically principled way of making rational decisions in the presence of uncertainty. There are different levels of complexity when it comes to MDPs. To give a simple example, let's talk about eating. Consider that it's late afternoon and you're hungry for a sandwich. Let's say you have three options: toss together whatever you can from the ingredients in your fridge, go to the store and buy the additional ingredients you need, or just go buy a sandwich from the deli. As you may already be able to tell, there's a lot of additional considerations that could be made to assist in your decision making. How much is the deli sandwich? How much are the ingredients? Will I use the remaining ingredients before they go bad? So, let's rank our objectives. What's most important to you? Is it to eat the yummiest sandwich? Is it to save money? Or is it just to make the overall most efficient decision based on these factors? To make the *best* decision, priorities must be established to define *best*. If saving money is most important, using up your fridge ingredients sounds like the way to go. If it's the best sandwich that you want, go buy that sandwich. As we can see, these decisions when laid out like this can quickly become overwhelming, not to mention the added complexity if you wanted to add even more detail to your decision model.

To better understand, let's break it down a touch further. MDPs can be split into two main types of processes: path and policy. Our sandwich example is more like a path type. It's a sequential series of events that's deterministic, meaning it produces the same result every time that the same sequential actions are made. You can create a *path* of actions that result in certain

outcomes. A path would be a *fixed* plan where as a policy would be a *reactive* plan. Within a fixed plan, we can know the exact outcome however steps into the future. On the other hand, the policy (reactive plan) can be thought of as a lookup table [17], storing what action to take for any given state. If I'm in state x, what's the best action to take? Instead of subjectively defining these best actions like we were able to in the sandwich example, in a policy MDP the best actions are calculated via the Bellman Equation. Simply put, it takes probabilities of different states and actions along with the possible reward of these and evaluates the best decision based on them. As the states and actions grow in complexity, it becomes near impossible for our human brains to determine this best decision reliably if at all.

So, how does this all transfer to RL? Well the point of RL is finding this "policy". The RL is the method we implement to figure out the policy. What we're essentially doing when training an agent is having the agent find the lookup table values itself. It's determining the policy on its own. We implement an algorithm to guide the agent in its learning process. RL is solving two problems simultaneously: First, it's trying to understand the consequences of the actions that it takes. Second, it's attempting to figure out what's the best policy. For the PPO RL algorithm specifically, PPO is learning two different neural networks (NN). It's trying to learn (1) the value function, and (2) the policy. Remember that while the value and transition functions may seem similar, but they are definitely not the same. The value function is trying to figure out how valuable (in terms of reward) a certain action is, given its current state. The transition function is trying to model the future state(s) based on the current state and action taken. The value function answers, "if I'm in this state and take this action, what is my expected reward?" while the transition function answers, "if I'm in this state and take this action, what will the next state be?" The MDP then is the combinatorial learning of the value function and policy.

3. Dataset

RL Datasets and Data Collection Overview - How Data is Being Generated

RL datasets are inherently different from those used for other types of ML techniques such as supervised and unsupervised learning. The data that RL models train on comes from the agent itself. RL dataset collection can usually be summed up using the acronym **SATR** which stands for **state**, **action**, **transition**, and **reward**. State represents the agent observing the current state of the environment and/or its position in relation to the environment. (after many observations, the agent may be able to begin correlating certain actions to their effect on the environment (depending on transition and reward functions)). Action represents the action that the agent takes in relation to the environment (do nothing, jump, move forward, etc). Transition represents the utilization of a transition function which predicts what the next state will be after the given action is taken. Finally, reward is really what incentivizes the agent. It represents the utilization of a reward function which rewards (and sometimes punishes) the agent based on how much a certain action determined the reward. In other words, the reward function learns how an

action influences how close the agent got to its goal. SATR is essentially the template of a policy. For all intensive purposes, you can think of all of these SATR parts combined into an algorithmic function as the full policy. Refer to Figure 2 to see an illustration of a SATR loop.

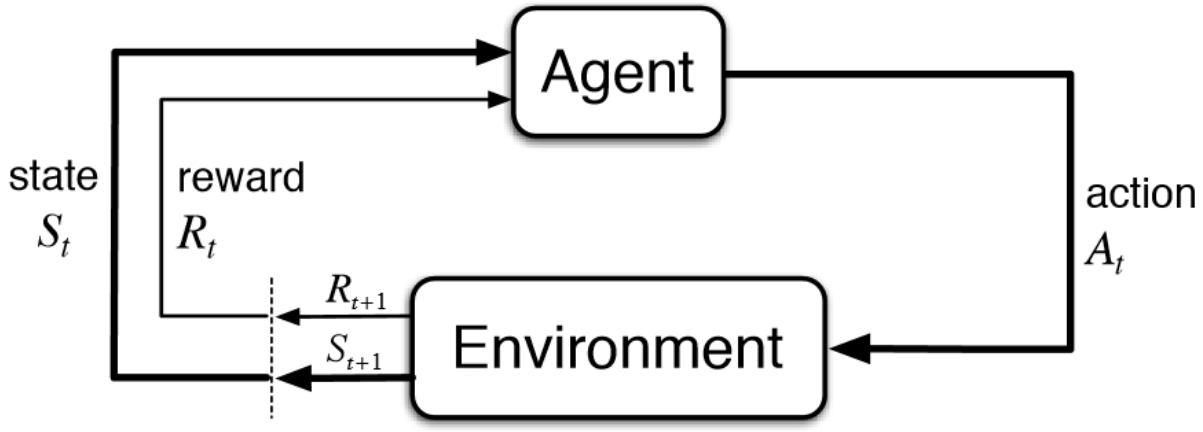

Figure 2: Image of the basic SATR looping process. The variable t represents a point in time during the training process, therefore $t + 1$ represents the *next* timestep. It's the time after time t. The agent starts with the current state (S_t) and then takes an action at current time (A_t) which has an effect on the environment. Then the next environment state (S_{t+1}) and associated reward (R_{t+1}) for taking that action are given back to the agent.

The CoinRun Environment - Where Data is Being Generated

While Procgen has 16 different retro style game environments [1], we only used CoinRun for this paper. CoinRun is a 2D platformer where the agent starts on the far left side of the level and has the goal of collecting the coin at the far right side (end) of the level [11]. Some levels have hostiles including moving enemies and stationary obstacles like spikes and lava pits. Refer to Figure 3 to see an example of a CoinRun level including hostiles.

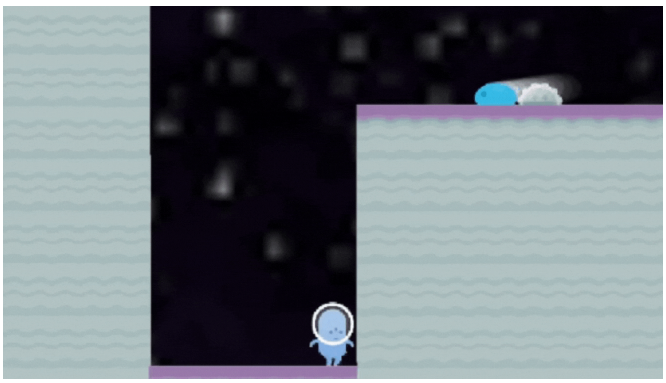

Figure 3: GIF of example of random CoinRun level including two player dangers and common performance of badly performing agent. Some enemies move while other dangers are stationary. This is a non-repeating random level generated when using 0 for the num_levels variable (num_levels=0). The agent behavior of walking into the wall continuously and other similar behavior was highly common in many of our agents.

Agent Data - <u>What</u> Data is Being Generated

What does the data and data collection for specifically our agents in Procgen CoinRun look like? Well, let's briefly revisit the SATR acronym and see how our specific Procgen CoinRun case correlates to it. Starting with the state, it's an image observation in the form of all of the red-green-blue (RGB) values of the CoinRun level that's within frame. That tells the agent the state, but whether or not the agent understands what those values mean or correlate to is part of what it's learning. Action in our case represents the action that the agent makes. It could really be anything within the range of movement allowed by the game. It could range from doing nothing to jumping while moving in a certain direction. The transition function is obviously unknown and evaluated by the simulator. If we knew the exact transition function, we wouldn't have to have the agent go through all these iterations to try to figure it out itself. Finally, the reward in our case represents the reward function evaluating how much a certain agent action influenced the amount of reward that it receives. Importantly though, an agent in our case only receives reward if it succeeds in the level (getting the coin). If it does anything other than reaching the coin, the reward is 0. If it reaches the coin and succeeds on the level, the reward is 10. There is no supplemental reward or negative reward, only none (0) and all (10). This all-or-nothing reward approach is known as "sparse reward". What was the actual data that we used for analysis? We collected and saved metric logs for all agents but focused on two key parameters for evaluating general performance: (1) num_levels, (2) total_timesteps. Initial and final reward were also heavily considered in the evaluation. There were 58 total agents that we trained and evaluated. Of these 58 total agents, 18 of them are considered the "Initial Set" or "Initial Agents". These 18 agents within the Initial Set trained on both a differing number of total timesteps (variable parameter name, total_timesteps) and a differing number of levels (variable parameter name, num_levels). The remaining 40 agents are called the "Experimental Set" or "Experimental Agents". These 40 agents in the Experimental Set were split into 4 subsets each consisting of 10 agents. All 40 Experimental Agents were trained on the same number of total timesteps, but each subset of 10 agents was trained on a different number of levels. Refer to Table 1 at the end of this section to see how agents are named and split up into sets and subsets.

Timesteps as a Variable Parameter and Its Influence on Metric Log Outputs

The total number of timesteps that the agent trains on determines how long the agent trains for, in other words, how many iterations the agent goes through. A metric log message is outputted to the terminal every 2,048 timesteps and the first (outputted after the first 2,048 steps) doesn't include training metrics, so it's best to use at least 4,098 timesteps. In this case, there would be two metric log message outputs. The first wouldn't have any training metrics, but the second would. These training metrics (which we didn't focus much on or change) included several metrics such as: entropy_loss, learning_rate, policy_gradient_loss, value_loss, and more. The base metrics outputted in all of the metric log messages including the first at 2,048 steps are what we mainly focused on and analyzed. These included rollout metrics (ep_len_mean and ep_rew_mean) and time metrics (fps, iterations, time_elapsed, and total_timesteps). The rollout

metrics specifically were our focus. Episode length mean (ep_len_mean) represents the average episode length. Our main priority though was the episode reward mean (ep_rew_mean) which represents the average reward received on episodes up to that point. Refer to Table 1 at the end of this section to see what number of timesteps are used for agent sets and subsets.

Number of Levels as a Variable Parameter

The number of levels determines the levels that the agent plays on. A value of 1 (num_levels=1) would result in the agent playing only on a specific pre-generated level. Refer to Figure 4 to see this default level. If the agent succeeds or dies on the level, it restarts that same level. A value of 5 would result in the agent playing through a set of 5 pre-generated levels including the same level as when num_levels is 1. Out of these 5 pre-generated levels, when the agent spawns, succeeds, or dies, the next level it plays is randomly selected from these 5 levels. If the agent doesn't succeed or die, and for example only jumps in place continuously, it would be limited to this level and not be exposed to any of the other levels in the 5 level selection. Because the next level is randomly selected, even if the agent succeeds or dies, it's possible for the agent to still not be exposed to all of the levels in the 5 level selection and continue to respawn in the same level(s). If num_levels is assigned a value of 0, the levels that the agent plays will never be the same or shown again. A value of 0 would result in the agent playing on levels from a selection of unlimited random ***non***-pre-generated levels. The values we used for our research were 0, 1, 3, 5, 10, 25, 50, and 100. Refer to Table 1 at the end of this section to see how the number of levels values are split up among agent sets and subsets.

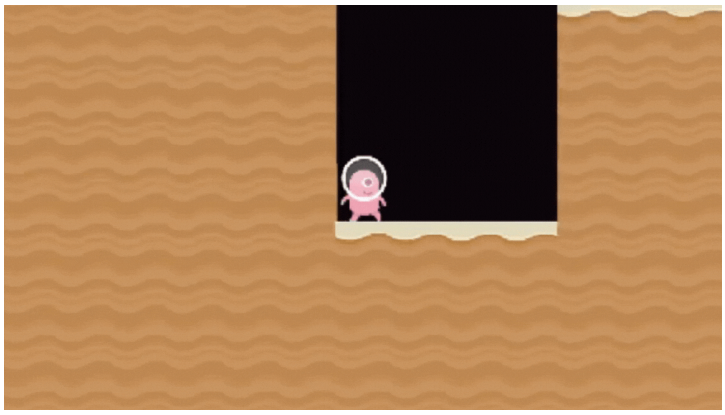

Figure 4: GIF of default level 1. This is the same level used repetitively for agents that train on the variable num_levels assigned a value of 1 (num_levels=1). Notice there are no hostiles or places for the agent to die. The coin is at the far right (end) of the level. The agent in this specific GIF is agent e_1_2 (refer to Table 1 below to see how agents are named and referred to).

Table 1: Table representing the name representations of agents and the splitting up of all 58 agents into two sets, the Initial set containing 18 agents and the Experimental Set containing the other 40 agents. Agents in the Experimental Set are further split into subsets of 10 agents where each subset of 10 trains on a different number of levels. Remember that 0 levels is NOT no levels, but is UNLIMITED levels. Levels used for 0 levels are non-repeating, random, and never reused or seen by the agent itself or any other agents again.

Agent Set Name	# of Agents	# of Timesteps	# of Levels	Agent Subsets	Agent Names
Initial Set	18	5,000 25,000 125,000	0 1 10 25 50 100	NONE All 18 Init. Agents	**i_1 - i_18** *set_agent#*
Experimental Set	40	25,000	0 1 3 5	**Subset 1** (1 Level) **Subset 2** (3 Levels) **Subset 3** (5 Levels) **Subset 4** (0 Levels)	**e_1_1 - e_1_10** **e_2_11 - e_2_20** **e_3_21 - e_3_30** **e_4_31 - e_4_40** *set_subset_agent#*

4. Methodology / Models

Brief Overview

We made a personal GitHub repository [4] for saving and sharing our work as well as using multiple other GitHubs from OpenAI (openai/gym [18], openai/gym3 [19], openai/train-procgen [20], openai/procgen [11]) and StableBaselines3 (DLR-RM/stable-baselines3 [21]) to aid in implementing Procgen at different stages from interactive personal gameplay, to implementing PPO with image inputs into our program. StableBaselines3 also helped us in implementing metric logging and saving using Tensorboard. Additionally, we used a Google Sheet spreadsheet to keep track of agent names, their log files, and which parameter variable values they were trained on.

Virtual Project Workspace Setup

We used GitHub as well as their desktop application [22] to save and share the code we used and additional project information. All of the Python [23] libraries and modules we used as well as all training and testing took place within a virtual environment and the code editor, Visual Studio Code (VSCode). We used a Google Sheets spreadsheet to keep track of results and other agent info. Refer to the rest of my GitHub Repository [4] for these spreadsheets and code files.

Using CoinRun Interactively

OpenAI's Procgen GitHub Repository, openai/procgen details how to run Procgen environments interactively [11], meaning playing the game yourself. Some packages and dependencies for interactive Procgen are not the same as what was used for RL in Procgen environments. We directly used gym3 which can be used for vectorized environments and imported the vectorized version of Procgen (ProcgenGym3Env) instead of the base version that we used for RL. The interactive version was used to play CoinRun interactively for the purpose of testing our Procgen implementation before using any RL. This allowed us to better understand CoinRun, its environment(s), and what actions agents were doing. The interactive CoinRun file and code we used is in my GitHub repository [4].

Procgen Implementation Packages and Dependencies

While we did look through multiple different Procgen related GitHubs including openai/procgen [11], openai/gym3 [19], openai/gym [18], and Farama-Foundation/Gymnasium (updated version of openai/gym) [24] in attempt to further understand dependencies of procgen, we found that they all somewhat used or were based on eachother. Gymnasium is an updated version of Gym, Gym3 is Gym with vectorization capabilities, and Procgen can use either Gym or Gym3, but usually uses both. So, we ended up directly using openai/procgen and importing the Python library, gym, which included everything we needed for the core Procgen environment. We only directly imported gym. However, env.py is a file in the Procgen package and GitHub which imports gym3 itself [4], [11]. As you may be able to see, the meshing of these packages was convoluted (especially between different GitHub repositories), but it was important to find these connections and give at least a brief summary of their conjunction here.

StableBaselines3 for PPO Implementation and Agent Metric Logging

StableBaselines3 is what we used for implementing PPO. Specifically, StableBaselines3's documentation [6] and GitHub (DLR-RM/stable-baselines3) [21] allowed us to very easily alter PPO for our use case. We changed only two parameters in the PPO function call. The first was the *Policy Network* (parameter name: "policy", the default value is "MlpPolicy" which is used for single-type input features other than images [8]). The second parameter was the Tensorboard log (parameter name: "tensorboard_log", the default value is "none" [9]). We changed the policy parameter from "MlpPolicy" to "CnnPolicy" which allows for the input data to be of image type, and since the observation space of the agent is all of the RGB pixel values within the frame, our input is indeed of image type. For the logging, we changed the tensorboard_log parameter from "none" to a file path where we stored the log files. Storing the log files allowed us to view graphs of certain training metrics for analysis and comparison later [7].

Training, Testing, and Automation

In order to expedite training, we automated our training code to train, log, and save agents sequentially without manual intervention by using a simple for loop. Refer to my GitHub repository [4] for this code file. The program for training a single agent starts by importing the PPO class from the StableBaselines3 module (stable_baselines3 in the code) and the gym module. It then creates the environment (env in code) by calling gym.make() with the procgen environment (procgen:procgen-coinrun-v0 in code), number of levels (num_levels in the code), and render mode (render_mode in code) as parameters. Next, the model is created by creating an instance of the PPO class via calling PPO with policy ("CnnPolicy" in code), the env (the env object previously created), verbose (an integer value parameter that determines which metrics are logged and printed to terminal, the default value is 1), and tensorboard log (tensorboard_log in code, defines the file location of where to store the log file(s)) as parameters. The learning starts by calling model.learn() with the number of total training timesteps (total_timesteps in code) as a parameter. After training is completed, the model is saved by calling model.save() with the file path and agent name combined into a string as a parameter. All it took to automate this process was putting everything after the imports into a for loop and defining a couple variables for naming the log file, naming the agent file, and changing the values assigned to the num_levels and total_timesteps parameters.

The test file for testing the trained agents is very similar to the training file. Refer to my GitHub repository [4] for this code file. It starts by importing the same modules (PPO class from stable_baselines3 module and gym module). Next, it creates the test environment (env_test in code) by calling gym.make() with the same parameters as the call in the test file. The model is then created by making an instance of PPO and calling PPO.load() with the file path and agent name combined string and the environment (the env_test object previously created) as parameters. Testing starts by converting the recently created model into a "vector environment" (vec_env in code) by calling model.get_env(). Then an observation variable (obs in code) is created and set equal to vec_env.reset() which returns an array of observations. A loop is then entered where we assign variables action and states (action, _states in code) to the method call model.predict() where the observation variable (obs) and deterministic boolean ("True" by default) are parameters. Still within this loop, the variables obs, reward, done, and info are assigned the value of the method call vec_env.step() which takes the action variable as a parameter. Finally (still within the loop), vec_env.render() is called to output the changed environment. The program stops once the loop has iterated through a number of timesteps or is manually halted. If the agent was trained on only 1 level, we would test it on that level once as well as one or two other random levels for comparison. We tested agents that were trained on more than 1 level until we saw its attempt at all of the levels that it trained on, with an exception for the agents trained on non-repeating random levels in which we tested each agent five times.

5. Results and Discussion

Varying the Number of Timesteps - Initial Set

On average, increasing the number of total timesteps in the Initial Set resulted in a higher mean reward at the end of training. Of the agents trained on 125,000 timesteps, the earliest that an agent achieved a mean reward of 10 was 67,584 timesteps (agent i_11). In fact, this was the only agent in the Initial Set to reach anything close to the max reward. Refer to Figure 5 to see this reward curve and the reward of agent i_11 over its training time. The next highest end mean reward achieved from an agent trained on 125,000 timesteps was a mean reward of 3.9 (agent i_12). The next best agent which had the same end mean reward of 3.9, was trained on only 25,000 timesteps (agent i_17). There were also two agents trained on only 5,000 timesteps that both had end mean rewards of 3.3. One of these agents which achieved the end mean reward of 3.3 (agent i_7) started at an initial mean reward of 5 and decreased over time. The other agent (agent i_10) had an initial mean reward of 4, then rose to 4.3 at 4,096 timesteps before decreasing. Some other agents that should be noted were agents i_13, i_14, and i_15. These agents were all trained on 125,000 timesteps. Agent i_13 achieved an end mean reward of 2.2, but had a drastic increase in mean reward starting at 65,536 timesteps. Before that point, its mean reward was consistently 0 or close to 0. Agent i_14 achieved an end mean reward of 3.7, but reached a mean reward of 4.3 at 45,056 timesteps and then decreased down to 2.9 at 108,544 timesteps and began recovering immediately after. Agent i_15 started at 0 and went to 3 at 26,624 timesteps before it decreased to 2.5 at 67,584 timesteps and then increased up to 3.4 at 116,736 timesteps before ending with an end mean reward of 3.1. Refer to Figure 6 for a reward graph of all 18 Initial Agents.

Already we can see that the agents who trained on more timesteps consistently ended with higher mean rewards, but also had the most fluctuation in mean reward over all timesteps. We can conclude that this was in part due to the inherent trait of timesteps, that is, the higher the number of timesteps, the more episodes and iterations the agent goes through. This results in more opportunity for change in mean reward. This is also why the graphs with higher numbers of timesteps are significantly more “curvy”. Many of the agents had a non-zero initial mean reward. However, other than agent i_18, 5 of the 6 agents that trained on 125,000 timesteps had an initial mean reward of 0, but 5 of the 6 agents (including agent i_18) also had the highest or greater than 3 end mean reward. That other agent (the remaining 1 out of 6) that didn’t have an end mean reward of 3 or greater (but did have an initial reward of 0), was agent i_13, the agent with the most drastic increase in mean reward. These results support the conclusion that generally, if given enough time, any agent would eventually achieve a consistent max mean reward of 10. These results are further supported by Figure 6.

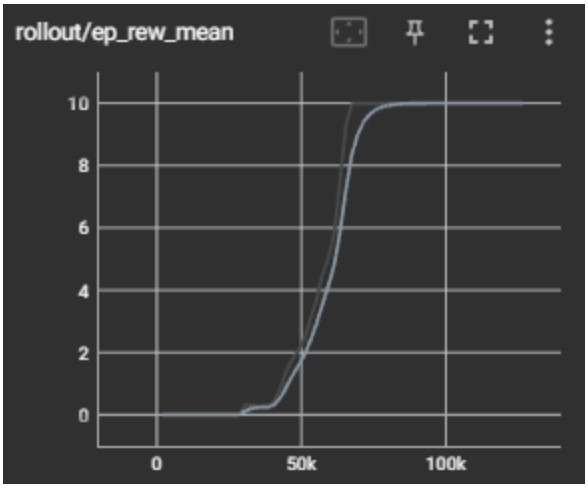

Figure 5: Mean reward graph for agent i_11. The x-axis represents the number of timesteps which in this case is 125,000 timesteps. The y-axis represents the mean reward per episode. Notice that nothing changes for the first few 10,000 timesteps and then reward quickly increases reaching the max of 10 around 65,000 timesteps.

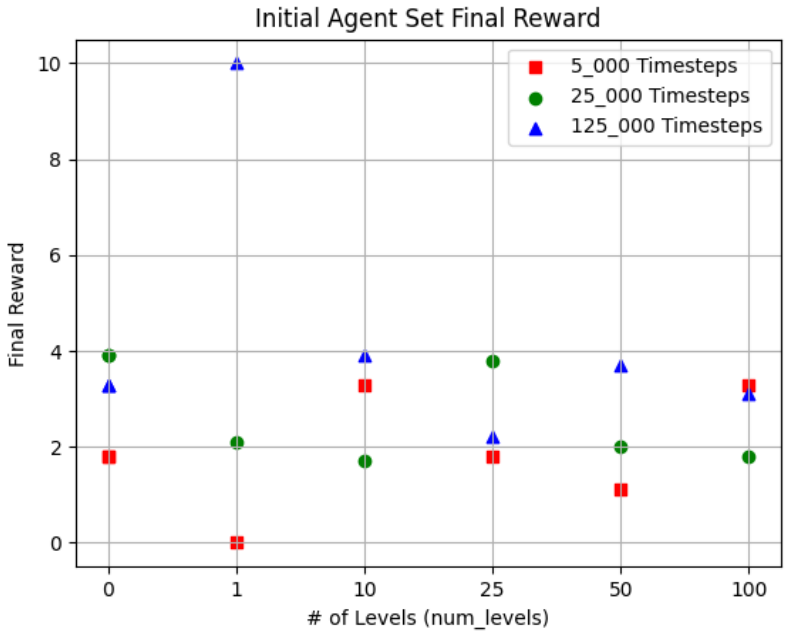

Figure 6: Scatterplot of number of levels by final reward where Initial Agents are colored and shaped differently according to the number of timesteps trained on. Remember that 0 levels (num_levels=0) does NOT mean no levels, but instead means unlimited non-repeating random levels. The agent trained on 125,000 timesteps and 1 level achieved the perfect max reward of 10. Notice that there is no discernable difference between performance for agents trained on 0, 10, 25, 50, or 100 levels. Agents that train on more than 1 level would most likely only reach the max reward if given significantly more timesteps to train on. This figure was made using Matplotlib [25].

Varying the Number of Levels - Experimental Set

The Experimental Agents were all trained on 25,000 timesteps, but were split into 4 groups of 10. Each subset of 10 agents trained on a different number of levels. The first subset trained on 1 level (num_levels equals 1) and included agents e_1_1 through e_1_10. The second subset trained on 3 levels (num_levels equals 3) and included agents e_2_11 through e_2_20. The third subset trained on 5 levels (num_levels equals 5) and included agents e_3_21 through e_3_30. Finally, the fourth and final subset consisted of agents e_4_31 through e_4_40 and were trained on non-repeating random levels (num_levels equals 0), where the number of levels parameter was assigned a value of 0. This information is also provided in Table 1. Refer to Figure 7 for histograms of all Experimental Agents by subset.

1 Level (Experimental Agents 1-10)

Overall, the agents that trained on 1 level had the highest end mean reward. In order, agents e_1_1 through e_1_10 achieved the following end mean rewards: 8.5, 10, 9.7, 10, 0.4, 1.7, 7.6, 0, 0, and 3.1. The average end mean reward between these 10 agents was exactly 5.1 ($\frac{8.5+10+9.7+10+0.4+1.7+7.6+0+0+3.1}{10} = 5.1$). Only four of these 10 agents had a non-zero initial reward. These were agents e_1_1, e_1_3, e_1_4, and e_1_10. The first, agent e_1_1, started with an initial reward of 5 and ended with an end mean reward of 8.5. Agent e_1_3 also started with 5, but ended with 9.7. Agent e_1_4 started with 6.7 and ended with 10. The tenth, agent e_1_10, started with 5 and ended with 3.1. One of the agents that started with an initial reward of 0 did end with an end mean reward of 10 though. This was agent e_1_2, the second agent. Two of the agents in this first subset of 10 agents both started and ended with mean rewards of 0. Refer to Figure 7 for a reward histogram of subset 1.

3 Level (Experimental Agents 11-20)

None of the second subset of Experimental Agents, those trained on 3 levels, ended with a mean reward of 10. In fact, none of them got above 8. The closest agent was e_2_12 which achieved an end mean reward of 7.3. The average end mean reward between all 10 agents in this second subset was 5.17 (~5.2) ($\frac{5.8+7.3+6.7+3.3+4.6+5.8+5.2+6.4+0.7+5.9}{10} = 5.17 \approx 5.2$) which is actually better than the first subset, but just barely. Even though 7.3 was the highest end mean reward that any agents of this second subset achieved, it was not the highest compared to initial reward. The third agent in this second subset, agent e_2_13, had an initial reward of 7.5, but then achieved an end mean reward of 6.7. Some other agents that had ended with a lower end mean reward than their initial reward were agents e_2_14 and e_2_19. The fourth, agent e_2_14, started with 4 and ended with 3.3. Agent e_2_19 started with 5 and ended with 0.7. While this was the closest end mean reward to 0, none of the agents in this second subset actually ended with an end mean reward of 0. Also within this second subset, 3/5 or 6 of the 10 agents had a non-zero initial reward. Refer to Figure 7 for a reward histogram of subset 2.

5 Level (Experimental Agents 21-30)

Like the second subset of agents trained on 3 levels, none of this third subset of agents that trained on 5 levels achieved a maximum end mean reward of 10. The highest end mean reward achieved by an agent in this subset was 6.3 and was achieved by the fourth agent in this subset, agent e_3_24. The next highest was very close at 6.2 achieved by agent e_3_28. Again like the second subset, no agents in this third subset ended with an end mean reward of 0. The lowest end mean reward was 2.4 achieved by the ninth agent, e_3_29. The next lowest was 4.2 achieved by agent e_3_23. The average end mean reward between all 10 agents in this third subset was 5.07 (~5.1) ($\frac{5+5.6+4.2+6.3+6+4.5+4.6+6.2+2.4+5.9}{10} = 5.07 \approx 5.1$). Of the agents in this third subset, 3/5 or 6 out of 10 agents had a non-zero initial reward. Those were agents e_3_21, e_3_22, e_3_25, and e_3_26, all of which ended with end mean rewards of 4.5 or higher. 6.7 was the highest initial reward and 3 of the 10 agents started with it. These were the agents e_3_23, e_3_29, and e_3_30. The ninth agent, e_3_29, was also the agent that decreased the most between its initial reward and end mean reward. It went from 6.7 to an end mean reward of 2.4. Agent e_3_25 had the largest increase between its initial reward of 0 and end mean reward of 6. Refer to Figure 7 for a reward histogram of subset 3.

0 Level (Experimental Agents 31-40)

Agents trained on non-repeating random levels once again had similar results as the agents trained on 3 and 5 levels. Within this fourth and final subset, the sixth agent, e_4_36, achieved the highest end mean reward which was 4. It did however rise up to 4.7 at 8,192 timesteps. This agent was also one of the 4 of 10 agents in the fourth subset that had an initial reward of 0. The other agents that started with an initial reward of 0 were agents e_4_33, e_4_34, and e_4_38. The next lowest initial reward was 2.5 which agents e_4_1, e_4_5, and e_4_10 started with. The seventh agent, e_4_37, had the highest initial reward in this subset which was 5. It ended with an end mean reward of 3.5. Agent e_4_39 had an initial reward of 4 and an end mean reward of 2.9, but it reached a mean reward of 6.3 at 6,144 timesteps. The average end mean reward between this last subset of 10 agents was 2.76 (~2.8) ($\frac{3.8+2+0.7+2.7+1.5+4+3.5+2.9+2.9+3.6}{10} = 2.76 \approx 2.8$), clearly significantly lower than the other 3 subsets of Experimental Agents. Refer to Figure 7 for a reward histogram of subset 4.

Comparing all 40 Agents in the Experimental Set

When we compare the results of all 40 Experimental Agents, we find some interesting and unexpected results. Refer to Figure 7 for reward histograms of each agent subset for additional comparison. Let's first compare the averages and highest end mean rewards of the 4 subsets of 10 agents. To recap, subset 1 agents, e_1_1 through e_1_10, which trained on only 1 level, had an average reward of exactly 5.10 and a maximum end mean reward of 10 achieved by 2, almost 3 of the 10 agents. Subset 2 agents, e_2_11 through e_2_20, which trained on 3 levels,

had a slightly higher average of 5.17 rounded to 5.2 and had a top end mean reward of 7.3. Subset 3 agents, e_3_21 through e_3_30, which trained on 5 levels, also had a near identical average 5.07 rounded to 5.1. The highest end mean reward achieved by an agent in the third subset was 6.3. The last subset of agents, e_4_31 through e_4_40, those that trained on non-repeating random levels, had a much lower average of 2.76 rounded to 2.8 and had a top end mean reward of just 4. Comparing agents in subset 1 with the other three subsets, it's important to note that agents in subset 1 were the only agents to reach the maximum end mean reward of 10 and in fact, the only ones to achieve and end mean reward greater than 7.3 (the highest achieved by agent e_2_12 of the second subset). This is most likely because of the fact that the agents trained on 1 level were only ever exposed to that exact same 1 level and were forced to either act until success or attempt success until time ran out. Not to mention that the specific 1 level played doesn't have anything that causes the agent to lose, such as enemies or lava pools. If half of these agents in subset 1 achieved an end mean reward higher than the next highest achieved by any other agents in the experimental set, it seems odd that its average is so similar to the other subsets (at least subsets 2 and 3).

One aspect that contributed to the decrease in the average end mean reward between all agents in subset 1, was the fact that two agents in this subset ended with a mean reward of 0. That combined with the fact that two of the other agents in this subset ended with 0.4 and 1.7 which correlates to the average of this subset being so close to the middle reward of 5. This also displays the extent to which randomness plays into agent performance, even when put into the simplest training environment. The same line of reasoning, that is only being exposed to the same level until time runs out and essentially being incentivised to memorize what worked once, can also be used to draw conclusions relating to the fourth subset of agents (agents 31 through 40), those trained on non-repeating random levels. That is to say, if an agent is never given the opportunity to test multiple techniques in a non-varying environment, it makes it significantly more difficult to learn what works and what doesn't. While this type of non-repeating exposure incentivises the development of more general skills and gameplay techniques, it also makes it significantly less likely for consistency to be built in a limited amount of time. One analytic that clearly conveys this inverse relationship between number of levels trained on and the time it takes to reach consistent successful performance is the pattern of decreasing top end mean reward of agents in each set and the increase in the number of levels trained on. For 1 level, the highest end mean reward was the max of 10. For 3 levels, the highest was 7.3. For 5 levels, it was 6.3. For the last subset, agents e_4_31 through e_4_40 who never saw the same level twice, it was the lowest at 4. We can also see that the initial reward had little to no effect on the end mean reward. In many cases where an agent started with a high initial reward, their end mean reward was actually lower than their initial. In fact, this was true for the majority of non-zero initial rewards. Out of all 40 agents within all 4 subsets, 18 of them started with an initial reward of 0. Out of the other 22, 10 agents ended with a higher end mean reward than their initial, and 12 agents ended with a lower end mean reward than their initial. Going slightly deeper, there was a 3:1 ratio of ending higher (3 ended higher than initial, 1 ended lower than initial) in the first

subset of 10 agents, a 3:3 ratio in the second subset of 10 agents, and a 2:4 ratio in both the third and fourth subsets. So we may draw a loose correlation between training on more levels, having a non-zero initial reward, and ending with a lower end mean reward than the initial. Based on our experimental set of 40 agents alone, there's roughly a 54.5% ($\frac{22}{12} \cdot 100 = 54.54\%$) chance that if an agent starts with an initial reward and is randomly selected to train on 1, 3, 5, or random levels, the agent's end mean reward will be lower than its initial reward. It could be interesting to see what the ratio would be if we trained agents on 1 level, but that 1 level is not the default level 1 and instead has dangers and enemies. It could also be informative to check if the pattern continues if we trained a set of 10 agents on 2, 4, or more levels. No matter what, more data, whether that be a larger number of agents per subset, a larger number of agent subsets overall, changing the default level for level 1 to be more challenging, or all of those and more together, is needed to draw more confident conclusions.

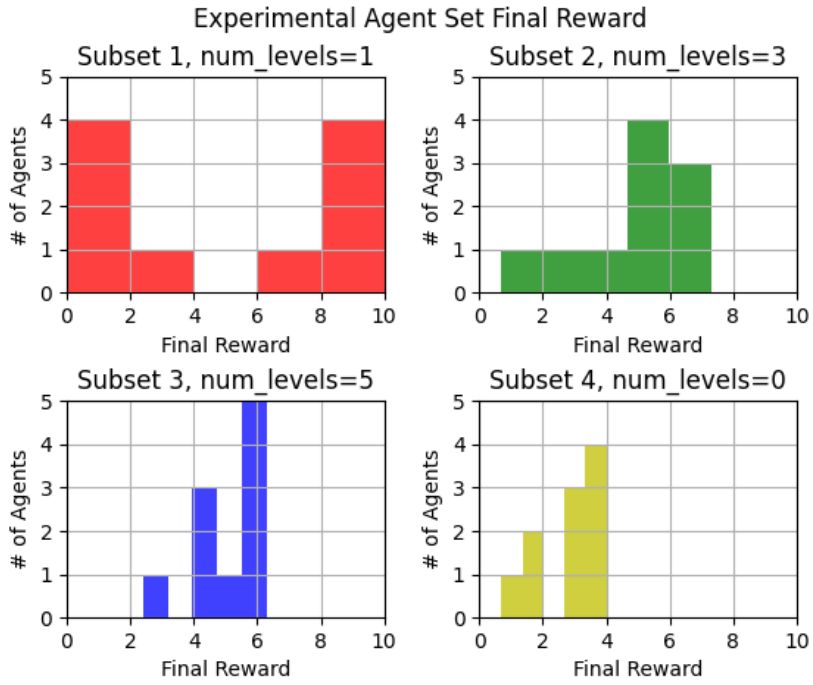

Figure 7: Combined histograms of final reward by number of agents within correlated Experimental Agent subset that achieved that reward. The top left (red) is Subset 1 containing agents e_1_1 through e_1_10 which trained on only 1 level (num_levels=1) for 25,000 timesteps. The top right (green) is Subset 2 containing agents e_2_11 through e_2_20 which trained on 3 levels (num_levels=3) for 25,000 timesteps. The bottom left (blue) is Subset 3 containing agents e_3_21 through e_3_30 which trained on 5 levels (num_levels=5) for 25,000 timesteps. . The bottom right (yellow) is Subset 4 containing agents e_4_31 through e_4_40 which trained on unlimited non-repeating random levels (num_levels=0) for 25,000 timesteps. Notice that in Subset 1, there was roughly the same number of agents that ended with low end rewards as there were that achieved high end rewards. This gives clarity on why the average reward between all Subset 1 agents was so close to a middle reward of 5. This figure was made using Matplotlib [25].

Analyzing Both Sets of Agents via Number of Timesteps and Levels

Now that we've compared both the Initial Set and Experimental Set individually, how do these sets compare with each other? Remember that the Initial Set (18 agents total) was trained on both a differing number of timesteps and levels. We saw that overall, the agents that trained for longer or a larger number of timesteps did better overall. The Experimental Set consisted of 40 agents total and was split into 4 subsets of 10 where all 40 agents were trained on 25,000 timesteps, but each subset of 10 agents trained on a different number of levels. Recall that the agents that trained on only 1 level had the highest end mean reward scores for individual agents of any set, but the average end mean reward between all 10 agents in subsets 1, 2, and 3 were very similar. So, we know that **more timesteps results in more consistent success**, and that **being exposed to fewer levels tends to result in a higher probability of consistent success**. Based on these two axioms, it makes sense that the best performing agent overall between both the initial and experimental sets was agent i_11 which trained on the highest number of timesteps we tested (125,000), and trained on only 1 level. Refer to Figure 4 to see this level. Once this agent had a technique that worked, it was able to double down on that technique because it didn't need to consider the generalization of that technique relative to other levels. Additionally, the level that's used when the number of levels is equal to 1 is a level without any dangers. It was essentially thrown into the level and forced to play until it ran out of time or succeeded. Then after it succeeded once, it had no incentive to further change or generalize the technique it used to succeed. So, not only did this agent have 5x more training time (timesteps) than any agents in the Experimental Set, but it also was able to spend 100% of that training time on essentially memorizing a single technique without regard to generalizing its skillset.

These results also align with those presented in Open AI's Procgen Benchmark. Agents that were trained on less than 10,000 levels tended to have immense difficulty in developing generalizable gameplay techniques. But, also as OpenAI showed in the article, when given enough time (a large enough number of timesteps) and a training set of 10,000 or more levels, an agent tends to develop generalizable techniques needed for consistent success. It's also important to remember the major difference in training time and timesteps used for our research compared to what OpenAI used for their benchmark. The highest number of timesteps that we trained on was 125,000 compared to OpenAI, which used 200,000,000 (200 million) timesteps—a vast difference of roughly 1600x ($\frac{200{,}000{,}000}{125{,}000} = 1600$).

Ways to Potentially Improve Training Reliability and Performance

Throughout the process of agent testing and evaluation, there were modifications to the training that we identified that could potentially increase the performance of agents trained in this low-intensity manner. First, let's consider the sparse reward problem. That is, the agent receives a max reward of 10 if it succeeds and a reward of 0 in all other cases. This greatly increases the reliance on randomly making actions that result in success, which in turn results in training and agent performance being significantly less predictable. One potential solution for

this would be implementing additional reward for something like the agent's distance from the coin. We could also implement negative reward, a punishment, for something like landing in lava or on an enemy. Ideally, we would take a step-by-step approach to testing out this additional reward, implementing a small change to how or when the reward is given and then looking for outcomes like more consistent and predictable agent performance, time it takes for different agents to solve the same level, and/or changes in movement patterns from different agents. Instead of changing the reward, we could also change what the agent sees in its observations. Currently, the agent sees only the RGB pixel value representations of the section of the agent's environment that's within frame. If we were to also give it the location of something like the coin or dangers (such as lava or enemies), it may increase performance, but at the very least would give us more data to analyze and potentially more insight into the factors most influential to consistent successful performance.

6. Conclusions

The emerging performance that we saw from many of our agents was significantly better than expected, especially because our focus was on low-intensity training. Multiple agents were able to display a mastery of at least one very basic level, if not multiple. These agents were able to achieve the maximum end reward of 10 even when trained on only 25,000 timesteps. Although low-intensity training does not *reliably* produce well performing agents on a consistent basis, it definitely *can* produce them. Our study didn't produce any on a consistent basis, but the consistency was better than what we expected to see. It's honestly incredible that any agents trained at this level were able to succeed at the rates they did. While we can conclude that high-intensity-training (or at least *higher*) is necessary to produce agents that can reliably generalize to new environments, the performance that was shown by our agents emphasizes the effectiveness of the PPO algorithm.

Through the analysis of our agents trained with this low-intensity training in mind, we identified two distinct trends. Agents trained on fewer levels tended to reach the max reward faster, but also tended to perform worse overall when tested on levels they weren't trained on. This gives us two positive relationships. First, there is a positive relationship between increasing the number of levels an agent trains on and the amount of time it takes for that agent to succeed consistently on those levels. Second, there is also a positive relationship between increasing the number of levels an agent trains on and that agent's general success on unseen levels. Although we didn't have enough data to conclusively evaluate whether or not these relationships are 1:1, based on the data we *do* have, and that given in the Procgen Benchmark, we can be fairly confident in saying that they are not 1:1 relationships, but definitely positively correlated.

Another key takeaway was that the initial performance and reward of an agent had little to no effect on long term and end training performance and reward. If anything, agents who had initial reward were more likely to have worse ending performance and reward. The nature of RL supports this, in that if an agent already has the lowest possible reward, its reward can really only

increase or stay as it is. Agents are free to try anything within their arsenal to increase reward. The worst that can happen is they don't improve at all. Though, if an agent starts with a reward of 5 for example and that agent were to try everything in *its* arsenal, all approaches that have a negative impact on it decrease the agent's reward. Simply put, an agent that starts with a reward of 0 will always either improve or remain unchanged, but an agent that starts with a non-zero reward has a higher chance of ending with less reward than it started with simply because 0 is the minimum.

Even though our low-intensity training was not successful at generalizing agent techniques, the Procgen Benchmark proves that there is in fact a big opportunity here for AI. RL models that are able to generalize to new environments have the potential to lay the groundwork for future AI advances. What would the results be if we were to develop AI with the trait of being able to generalize to new datasets (not just environments) at its core? Remember that one of the, if not *the*, biggest takeaway from the Procgen Benchmark was that agents would continue to improve when given more and more levels. This is a rare trait for any AI model because normally they fall victim to overfitting—the process of performing too close to the dataset the model was trained on. It's practically the opposite of generalization. If AI was developed with a generalization motive at its core, the results could help overcome the challenge of overfitting. Not only could datasets be more general **and larger**, but the AI model training on this dataset may be better equipped to learn more from individual data components in the set as well as have the info it's learning be less specified and therefore, more generally applicable. Additionally, this approach may result in biased data having less of an effect on the model's learning overall. This is due to the presence of more data in the dataset. Subsequently, the biased data becomes a smaller fraction of the overall pie making it carry less influence. The model could also be better protected against biased data influence simply due to the fact that its goal now is not to mimic its training data, but instead to gain a general understanding of its training data and what aspects of it can be further generalized. More specifically, it's goal would now be to learn info that develops and supports generally applicable skills. While this may sound unrealistic or at least easier said than done, I believe that this is actually the direction AI is moving toward. I also believe that the more we explore various creative approaches to AI development while keeping an open mind, the better the results and their impact on our future.

A Message for the AI Industry - Addressing the Big Picture

I'd like to end by panning out and talking about arguably the largest current problem in AI development which is being ignored by a vast portion of the industry. In January 2023, a human Go amateur defeated the world's current best AI Go model, KataGo, with a win rate of over 93%. The human won 14 of the 15 games played against KataGo. So, why is this a problem and why am I explaining it here? At first glance, this news sounds the same as many of the AI headlines that come out what feels like every other day now. However, this result highlights a fundamental flaw with our current AI development techniques, so fundamental that it undermines any models being built on that existing framework and with those same methods.

The impact that it could have if not addressed in the very near future could be devastating to both our world, which is increasing its reliance on these systems every day, and our relationship with AI as a whole.

Kyle Hill does an outstanding job explaining the game of Go, what happened with KataGo, and the major problem we must face. “Go is an ancient game where players take turns placing black and white stones on a board [wooden 19 x 19 grid board] with interconnected spots that allow adjacent stones to become so-called *groups*. The player that has the most occupied area or captured groups of an opponent’s stones at the end wins after all the stones are placed. Up until 2023, no human alive was better than the best AI at this, and then researchers from MIT and UC Berkeley made a superhuman AI look… like a toddler” [26]. The researchers identified some weaknesses of KataGo and proceeded to create an adversary model which utilized these weaknesses to play against KataGo. The adversary model used the basic strategy of systematically surrounding KataGo’s pieces in what the researchers called a “Double Sandwich method” [27] basically, “a surround of a surround and so on” [26]. “The Double Sandwich technique was so successful in fact that it wasn’t just winning, it was making it seem like the superhuman AI **didn’t actually understand what a group of stones was**, a fundamental part of the game” [26]. It’s important to note that the adversary wasn’t playing better Go than KataGo, but instead simply “taking advantage of a fundamental lack of understanding. That understanding being that groups of stones are… groups of stones, that need to be protected” [26]. The amateur Go player was one of these researchers, Kellin Pelrine [27]. He learned the Double Sandwich technique and used it to beat KataGo in 14 out of 15 games. When faced with a challenge meant to test its true understanding of the fundamental concept of groups of stones, KataGo failed almost completely. “The critical takeaway from this is not just that GoBots have hidden flaws. It’s that we don't actually know how today’s most widely used AI systems work, even the ones we thought were essentially solved” [26].

KataGo uses the same basic architecture and method of learning that all of today’s most popular AI systems like ChatGPT use, and it allows them to do incredible things. “They can diagnose rare medical conditions in milliseconds better than any doctor. They can pass the Bar Exam better and faster than any lawyer. But, **no one has ever shown that these systems actually understand anything at a fundamental or conceptual level**. They can write screenplays, but don’t know that screens exist. **They can play Go, the worlds’ oldest board game, but don’t know that boards are where you play that game**” [26]. This is the problem at hand. “Right now, the intelligence these systems mimic is impressive, but now it’s just that, **mimicry**. Deep learning systems like ChatGPT are using big data to approximate the world, **not develop any understanding of it**. And because of that, they can make **mistakes that even a calculator wouldn’t**. They can be abused and have Achilles heels we won’t expect. Right now, the solution to making these systems better is to just give them more data, **rather than understand them from the ground up**. This isn’t systematic and doesn’t add any more transparency or conceptual understanding to the AI themselves” [26]. If this is the way our current most advanced *and influential* AI models are being developed, what’s the magnitude of

the threat this poses to our modern world? What even is the threat? “If these systems get slotted into society at large, we won’t even know what specifically is going wrong if **and when** it does. Large language models are going to give people bad medical advice on Google.Combined with deep fakes, bad actors are going to create an **exponentially expanding arsenal of propaganda and misinformation** for almost no cost on social media. The fabric of democratic society is going to tear when it gets to **the point where no one knows whether or not anything is real**. Top AI researchers voice these concerns all the time, but it **doesn’t seem like anyone is listening**” [26]. So not only have professionals understood this flaw and vented its risks for years now, but the people currently in positions of power best equipped to tackle this problem are actively ignoring the issue. This is very similar to the insufficient response to climate change realities. What’s the next step then, and what can we expect to happen if we don’t take action soon? “We’ve been far too hasty in ascribing superhuman abilities to these machines as one of the researchers [Stuart Russell] put it, and this is going to **inevitably lead to unforeseen consequences** be they social, political, economic, or ethical. Perhaps it is time to step back. Perhaps it is time to stop rushing head first into AI that are essentially **alien** to us. ChatGPT is the single fastest growing consumer application in human history, and **we don’t fully understand how it works**. If AI really is the future, is this how we want to get there” [26]?

Even something like the Procgen Benchmark that was made to study the generalization of RL agents is not at all fundamentally different from these super mimicry models. While the training techniques they used were meant to incentivize generalization, the learning technique itself is still PPO which is still made using the same techniques as other popular learning algorithms. If you train a superhuman Chess model but then test it on a game of Chess where you change the rules, say half the pieces have different movements from traditional Chess, the superhuman Chess model probably wouldn’t perform very well. It’s not learning how to play the game or even understand games in general, it’s just mimicking all the gameplay iterations that it’s seen. Procgen Benchmark agents are doing the exact same. This won’t change until we **innovate the fundamental development techniques we use for AI**. This is not the time to fear ridicule from colleagues for trying something new. It is not the time to value profits over people. We need to work together and share solutions. If there’s anything we learned from the popularity of ChatGPT, it’s that collaboration is the most effective way of identifying and solving big issues. Even if you’re the smartest person in the world, you will **always** be outperformed by collaboration. The world we live in is a result of collaboration, and its flaws and vulnerabilities stem from competition. Stop competing, start collaborating, and take action **now.**

Acknowledgments

Jacob Le Mons (Jake Lemons) was supported and mentored by Joseph (Joe) Vincent who provided continuous guidance throughout the research and writing processes. Joe contributed both conceptual and technical knowledge and experience. *Thank you, Joe. I couldn’t imagine being able to accomplish or do this without you.*

Jake was also supported and given this opportunity by InspiritAI who've been providing their resources and learning content to Jake throughout multiple different AI learning programs since 2021.

References

[1] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, "Procgen Benchmark," December 19, 2019, https://openai.com/research/procgen-benchmark

[2] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, "Leveraging Procedural Generation to Benchmark Reinforcement Learning," pp. 1-19, 2019, https://arxiv.org/pdf/1912.01588.pdf

[3] J. Schulman, O. Klimov, F. Wolski, P. Dhariwal, A. Radford, "Proximal Policy Optimization," July 20, 2017, https://openai.com/research/openai-baselines-ppo

[4] lemonsjake/Low-Intensity-RL-Procgen-PPO, "Low-Intensity Reinforcement Learning Research Project Repository," last updated May 9, 2023, https://github.com/lemonsjake/Low-Intensity-RL-Procgen-PPO

[5] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," Journal of Machine Learning Research, vol. 22, no. 268, pp. 1-8, 2021, https://jmlr.org/papers/volume22/20-1364/20-1364.pdf

[6] Stable Baselines3, "Stable Baselines3 Docs - Reliable Reinforcement Learning Implementations," https://stable-baselines3.readthedocs.io/en/master/

[7] Stable Baselines3, "Tensorboard Integration," 2.0.0a8 Documentation, https://stable-baselines3.readthedocs.io/en/master/guide/tensorboard.html

[8] Stable Baselines3, "Policy Networks," 2.0.0a8 Documentation, https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html

[9] Stable Baselines3, "PPO," 2.0.0a8 Documentation, https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

[10] Farama Foundation, "Gymnasium Documentation," v0.28.1, https://gymnasium.farama.org/

[11] openai/procgen, "Procgen Benchmark," last updated January 24, 2022, https://github.com/openai/procgen

[12] The AlphaStar Team, "AlphaStar: Mastering the real-time strategy game StarCraft II," January 24, 2019, https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii

[13] D. J. Wu, "Accelerating Self-Play Learning in Go," pp. 1-28, November 10, 2020, https://arxiv.org/pdf/1902.10565.pdf

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, "Proximal Policy Optimization Algorithms," pp. 1-12, 2017, https://arxiv.org/pdf/1707.06347v2.pdf

[15] C. Chun-Yu, Y. Ting-Guang, P. Yun-Chung, W. Chen-Hua, "Trust-region methods," last updated 01:18, December 16, 2021, https://optimization.cbe.cornell.edu/index.php?title=Trust-region_methods

[16] TensorFlow, “Playing Cart-Pole with the Actor-Critic method,” last updated December 20, 2022, https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic#:~:text=Actor%2DCritic%20methods&text=In%20the%20Actor%2DCritic%20method,based%20on%20the%20given%20policy
[17] Hawes, Nick “Markov Decision Processes - Computerphile,” YouTube video, October 25, 2022, https://www.youtube.com/watch?v=2iF9PRriA7w
[18] openai/gym, “Gym,” last updated January 30, 2023, https://github.com/openai/gym
[19] openai/gym3, “gym3,” last updated January 1, 2021, https://github.com/openai/gym3
[20] openai/train-procgen, “Leveraging Procedural Generation to Benchmark Reinforcement Learning,” last updated June 10, 2020, https://github.com/openai/train-procgen
[21] DLR-RM/stable-baselines3, “Stable Baselines3,” last updated May 8, 2023, https://github.com/DLR-RM/stable-baselines3
[22] GitHub, “GitHub Desktop,” 2023, https://desktop.github.com/
[23] Python, “Python Documentation,” latest release (stable 3.11.3), https://docs.python.org/3/
[24] Farama-Foundation/Gymnasium, “Gymnasium,” last updated May 8, 2023, https://github.com/Farama-Foundation/Gymnasium
[25] Matplotlib Development Team, “Matplotlib Documentation,” v3.7.1, https://matplotlib.org/stable/
[26] Hill, Kyle “The HUGE Problem with ChatGPT,” YouTube video, April 24, 2023, https://www.youtube.com/watch?v=l7tWoPk25yU&t=806s
[27] T. Wang, A. Gleave, T. Tseng, N. Belrose, K. Perine, J. Miller, M. D. Dennis, Y. Duan, V. Pogrebniak, S. Levine, S. Russell, “Adversarial Policies Beat Superhuman Go AIs,” 2022, https://goattack.far.ai/#:~:text=We%20attack%20KataGo%2C%20a%20state,enough%20search%20to%20be%20superhuman