Generalizing PPO with Procgen

Alessandra Blasioli
Autonomous and Adaptive Systems Project
`alessandra.blasioli@studio.unibo.it`

Abstract

Procedural generation in reinforcement learning (RL) environments is a powerful tool to test agents' ability to *generalize* beyond memorized behaviors. In this project I apply Proximal Policy Optimization (PPO), a widely used on-policy RL algorithm, to a subset of the Procgen Benchmark, training agents in four visually and cognitively distinct games: *StarPilot*, *CaveFlyer*, *CoinRun*, and *BigFish*. This report details the network architectures, algorithmic decisions and the evaluation metrics, used to assess training stability and generalization.

1 Introduction

Generalization in RL remains challenging, particularly when agents are trained in environments where experiences repeat. The **Procgen Benchmark** [1] addresses this by generating new levels for each episode using procedural content generation. Sixteen such environments test both sample efficiency and generalization, making them ideal for evaluating RL algorithms under diverse conditions.

PPO [2] is a popular policy-gradient method thanks to its balance of sample efficiency and optimization stability. Prior studies have demonstrated PPO's performance on the complete Procgen suite. However, comparative analyses across diverse games, emphasizing per-environment behavior, remain valuable for understanding strengths and weaknesses.

This work implements and evaluates PPO in multiple Procgen environments.

2 Procgen Environments

The **Procgen Benchmark** is a collection of procedurally generated reinforcement learning environments that has a 15 dimensional action space and produce $64 \times 64 \times 3$ RGB image observation [1]. In this work, the focus is on four diverse environments that span different cognitive and motor challenges: **StarPilot**, **CaveFlyer**, **CoinRun**, and **BigFish**. Each game emphasizes different aspects of perception, planning, and control, making them suitable for evaluating robustness and generalization of policy learning. The environments can be summarized as follows:

- **StarPilot:** Top-down space shooter where the agent dodges projectiles and shoots enemies; requires fast reactions and spatial awareness in procedurally generated combat.
- **CaveFlyer:** Navigate a small ship through narrow caves, emphasizing precise thrust control to avoid collisions; a single mistake ends the episode.
- **CoinRun:** Side-scrolling platformer with procedurally generated levels; tests navigation, obstacle avoidance, and generalization to new layouts.
- **BigFish:** Underwater predator-prey environment; the agent balances risk and reward, eating smaller fish while avoiding larger predators, requiring long-term strategy.

Workshop on Instruction Tuning and Instruction Following at NeurIPS 2023.

3 Method

3.1 Architecture and Design Choises

I implemented separate convolutional neural networks (CNNs) for the actor and critic. The actor outputs action probabilities via softmax, while the critic produces a single scalar state value.

Initially, the actor network consisted of four fully connected layers ($256 \rightarrow 128 \rightarrow 64 \rightarrow 32$ units) with ReLU activations and dropout, followed by a softmax output layer. Experiments showed that this architecture struggled with visual input, and dropout did not improve performance—removing it slightly improved results. To overcome the limitations of dense layers for images, I switched to a convolutional design. The final actor network comprises three convolutional layers ($32\times8\times8$ stride 4, $64\times4\times4$ stride 2, $64\times3\times3$ stride 1), a flattening layer, a dense layer of 512 units, and a softmax output layer, enabling efficient spatial feature extraction and significantly improved performance.

For the critic, I initially used three fully connected layers ($256 \rightarrow 128 \rightarrow 64$ units) with ReLU and dropout, followed by a single scalar output. In the final version, the critic mirrors the actor's convolutional architecture: three convolutional layers ($32\times8\times8$ stride 4, $64\times4\times4$ stride 2, $64\times3\times3$ stride 1), a flattening layer, a 512-unit dense layer, and a final output producing a single value estimate. This design allows the critic to extract rich spatial features from visual input, improving value estimation accuracy.

I designed my actor and critic network architectures taking inspiration from the convolutional design of the original DQN by [3] and adapting it for use in my PPO implementation, with key modifications including a different output structure — the DQN uses a linear layer producing Q-values without softmax, whereas my actor outputs a softmax distribution over `num_actions` and my critic outputs a single scalar — and a change in input normalization, which in DQN is performed by dividing by 255.0 inside the network, while in my case it is applied during the `collect_rollout` stage.

3.2 PPO Algorithm

The on-policy updates are performed with the following parameters:

- **Generalized Advantage Estimation (GAE)**: $\gamma = 0.99$, $\lambda = 0.95$.
- **Clipped objective**: restricts policy updates within $[1-\epsilon, 1+\epsilon]$, $\epsilon = 0.2$.
- **Entropy bonus** ($\beta = 0.02$) to maintain exploration.

Rollouts of length 256 steps are collected per update, with minibatches of 2048 samples. Each rollout collects 256 steps from a single environment per update, producing a batch of transitions that is reused for several optimization epochs.

3.2.1 Advantages

Initially, a simple discounted sum of temporal differences was used to estimate advantages. To improve stability and control the bias–variance trade-off, **Generalized Advantage Estimation (GAE)** [4] was implemented. The advantage at time step t is computed as:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\,(1-d_t)\,\hat{A}_{t+1}, \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t).$$

Using λ allows the network to propagate advantage information across multiple time steps while still discounting distant rewards, improving learning stability in environments with long or delayed reward structures.

Before being used in the policy update, advantages are normalized to zero mean and unit variance to improve numerical stability:

$$\hat{A} \leftarrow \frac{\hat{A} - \mu(\hat{A})}{\sigma(\hat{A}) + 10^{-8}}.$$

Returns for the critic are computed as:

$$R_t = \hat{A}_t + V(s_t). \tag{1}$$

3.2.2 Policy Update: Full Batch vs. Minibatch PPO

Initially, the agent was trained using a **full-batch update** scheme, processing all experience tuples $(s_t, a_t, r_t, s_{t+1}, d_t)$ from a rollout in a single gradient step. The policy loss, following the clipped surrogate objective introduced in [2], is defined as:

$$L^{\text{policy}} = -\frac{1}{N} \sum_{t=1}^{N} \min\Big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\Big), \tag{2}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$ is the probability ratio between the updated and old policies.

In practice, full-batch updates exhibited high gradient variance and often led to stagnation near random-agent performance. To improve stability, the update scheme was reformulated as **minibatch PPO**, where:

- The collected rollout is shuffled to break temporal correlations.
- The data is split into minibatches of fixed size (2048 samples).
- Multiple optimization epochs (e.g., 4) are performed over the same rollout data.

This approach reduces gradient variance, enables multiple passes over the same data, and empirically results in smoother gradient updates and more consistent learning dynamics.

The policy and value networks are updated separately, each with its own optimizer. The total loss for the actor combines the clipped policy loss, the value loss (scaled by $c_v = 1.0$), and an entropy bonus term (scaled by $\beta = 0.02$) to encourage exploration:

$$L_{\text{total}} = L^{\text{policy}} + c_v \cdot L^{\text{value}} - \beta \cdot \mathbb{H}[\pi_\theta(\cdot|s_t)].$$

3.2.3 Trajectory Collection

The trajectory collection function (`collect_rollout`) interacts with the environment for a predefined number of steps, gathering all the information necessary for updating both the policy and the critic.

At each time step, the following information is recorded:

- **Observed state:** Normalized and stored in a buffer to ensure consistent numerical representation.
- **Chosen action:** Sampled from the actor's categorical distribution; stored for log-probability computation.
- **Received reward:** Used for advantage and return estimation.
- **State value:** Estimated by the critic; used in GAE and value loss computation.
- **Action log-probability:** Required for computing the PPO probability ratio.
- **Termination flag:** Indicates episode end for correct advantage calculation.

When an episode ends within the rollout, statistics such as total episode reward and length are logged for monitoring. At the end of the rollout, buffers are converted into arrays, and the value of the final next state is estimated to complete the GAE computation. Advantages and returns are then passed to the update phase.

3.3 Training Loop and Evaluation

PPO is trained sequentially per environment (*StarPilot, CaveFlyer, CoinRun, BigFish*) for 1000 updates. Each update collects a rollout, computes advantages and returns, and performs minibatch PPO updates on the actor and critic. Periodic evaluations measure mean and standard deviation of episode returns on held-out levels. Training and evaluation metrics are logged via Weights & Biases [5], including policy loss, value loss, entropy, advantage statistics, episode rewards, and lengths. Checkpoints and evaluation routines are separated by environment using unique W&B run names for clarity.

Although some curves in the training and evaluation plots appear shorter than others (e.g., *coinrun* and *caveflyer* compared to *starpilot*), this does not indicate that the corresponding runs terminated earlier. All experiments were trained for the same number of updates (1000). The apparent discrepancy is due to the logging strategy adopted: metrics are logged at the end of each episode, and the default `step axis` in W&B reflects the number of logged data points rather than the training updates. Since different environments generate episodes of varying average length, the logging frequency differs across games. Environments with shorter episodes (e.g., *coinrun*, *caveflyer*) produce more frequent logging events, resulting in fewer plotted points along the `step` axis. Conversely, environments with longer episodes (e.g., *starpilot*) generate fewer logging events, so their curves extend further along the x-axis. Importantly, all runs completed successfully and reached the same training horizon.

3.3.1 Experimental Setup

Experiments were conducted on two configurations of each Procgen environment:

- **Training:** initial experiments used 200 levels in *easy* mode, followed by 500 levels in *hard* mode to evaluate learning under more diverse and challenging conditions. Training always started from `start_level=0`, meaning the agent was exposed to a fixed set of levels throughout training. This setup facilitates learning by reducing variability, but limits diversity in the training distribution.
- **Evaluation:** for assessing generalization, episodes were run on both 200 and 500 levels, starting from `start_level=500` for the latter, ensuring the agent had not seen these levels during training. Each evaluation consisted of 10 episodes, with the agent acting deterministically by selecting the action with the highest probability from the policy network (i.e., using `argmax` over the policy output), in contrast to the stochastic sampling used during training.

Note: During evaluation, no rendering was performed to improve computational efficiency and reduce runtime. This setup allows comparison of both learning efficiency and the ability to generalize to unseen, procedurally generated levels.

3.4 Random Agent Baseline

To provide a reference for PPO performance, I implemented a **random agent** that follows the same interaction loop with the environment as the PPO agent, but selects actions uniformly at random.

The random agent:

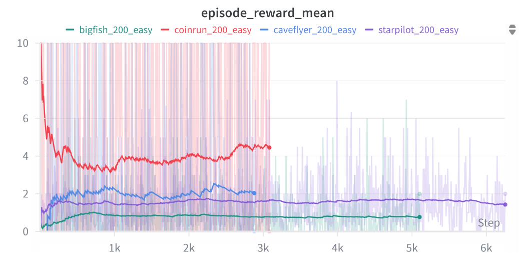
- Is initialized with the number of available actions.
- Chooses an action randomly at each time step from the action space.
- Does not require training, weight saving, or loading.

Statistics collection, evaluation, and logging are handled similarly to the PPO agent, with Weights & Biases used to track episode rewards, lengths, and aggregated metrics. This setup provides a **baseline** to quantify the advantage gained by the learned PPO policy over purely random behavior.

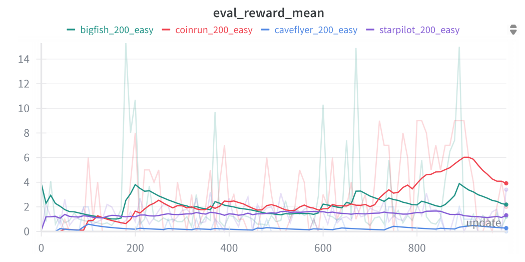
4 Experimental Observations

Among the environments tested, *CoinRun* in *easy* mode with 200 levels stands out as the only case where the agent consistently performs better than a random baseline. In this setting, the evaluation curve shows a modest but clear upward trend, rising from a mean reward of around 2 to over 4. However, in the remaining games under the same configuration (*StarPilot*, *CaveFlyer*, *BigFish*), the agent fails to demonstrate meaningful improvement, with noisy reward curves and no consistent learning signal.

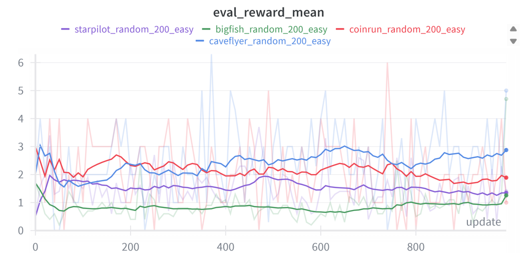
When training is extended to 500 levels in *hard* mode, performance deteriorates further. Even *CoinRun*, which previously showed signs of learning, does not outperform the random baseline in this more challenging setting. The evaluation curves across all environments remain erratic, and no clear improvement trend is observed.



(a) Episodic Reward (mean) of all the games played in easy mode, 200 levels. (x-axis is `step`)

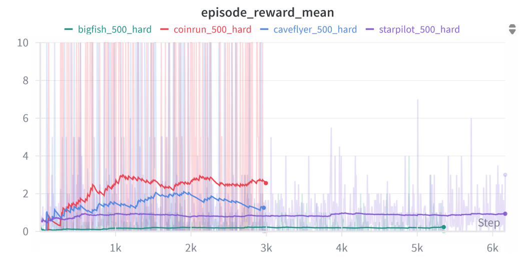


(b) Evaluation Reward (mean) of all the games played in easy mode, 200 levels.

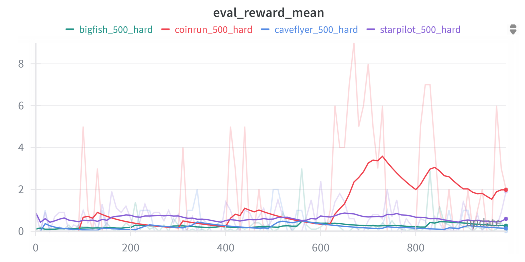


(c) Evaluation Reward (mean) of all the games played in easy mode, 200 levels, with the Random Agent.

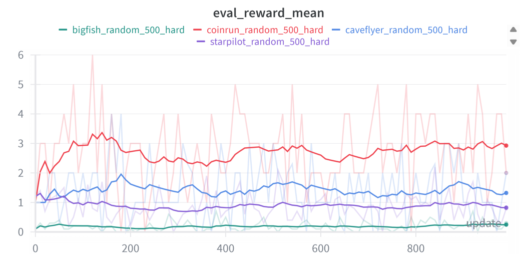
Figure 1: Rewards (mean) of all the games played in easy mode, 200 levels



(a) Episodic Reward (mean) of all the games played in hard mode, 500 levels. (x-axis is `step`)



(b) Evaluation Reward (mean) of all the games played in hard mode, 500 levels.



(c) Evaluation Reward (mean) of all the games played in hard mode, 500 levels, with the Random Agent.

Figure 2: Rewards (mean) of all the games played in hard mode, 500 levels

5 Conclusion and Future Work

While the agent did not achieve strong performance across all environments, the results highlight key limitations and offer valuable insights for future improvements. One clear constraint was the short training duration: **1000 updates** are insufficient for PPO to converge in complex, procedurally generated environments. This restriction was due to the limited time and resources available, which made longer training runs impractical.

In addition, the architecture used in this work is relatively shallow compared to those adopted in prior studies on Procgen. Most competitive implementations employ deeper convolutional stacks, residual connections, and normalization layers to better capture the visual complexity and variability of these environments.

The main bottlenecks identified are:

1. **Model capacity** — the architecture lacks the representational power needed for complex visual tasks.
2. **Training time** — the number of updates is too low to allow stable policy improvement.

Future work should address both limitations by exploring more sophisticated PPO implementations, such as those using deeper or residual CNN backbones, and significantly extending the training horizon. Additional enhancements — including improved exploration strategies, regularization techniques, and data augmentation — may further boost generalization in procedurally generated environments.

In conclusion, although the current results are modest, this work has laid a solid foundation for future research and has deepened my understanding of reinforcement learning in visually rich, stochastic domains.

6 Code Availability

The source code used for this work is publicly available at the following GitHub repository:

`https://github.com/alessandrablasioli/`
`AAS-Exam-Project-Generalizing-PPO-with-Procgen`

References

[1] Cobbe, K., Hesse, C., Hilton, J., & Schulman, J. Leveraging Procedural Generation to Benchmark Reinforcement Learning. *arXiv preprint arXiv:1912.01588*, 2019.

[2] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.:contentReference[oaicite:0]index=0

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. `https://www.nature.com/articles/nature14236`, Accessed 2025.

[4] Schulman, John, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[5] Weights & Biases. Weights & Biases Documentation. `https://wandb.ai/`, Accessed 2025.