Autonomous and Adaptive Systems Project Report

Giacomo Gaiani
Alma Mater Studiorum - University of Bologna
giacomo.gaiani@studio.unibo.it
https://github.com/GiacomoGaiani

Abstract

This project focuses on the development of a reinforcement learning (RL) agent capable of mastering a subset of games within the Procgen Benchmark, a suite designed to evaluate both sample efficiency and generalization across procedurally generated environments. Leveraging Proximal Policy Optimization (PPO), the agent learns to balance exploration and exploitation while interacting with diverse, procedurally generated game environments.

1 Introduction

In recent years, reinforcement learning (RL) has emerged as a powerful framework for solving complex decision-making problems, particularly in environments that demand both exploration and exploitation. One prominent challenge in RL research is developing agents that not only perform well in known environments but can also generalize to new, unseen scenarios. The Procgen Benchmark, a suite of procedurally generated environments, provides a rigorous platform for testing the generalization capabilities of RL agents. Unlike fixed game environments, Procgen games continuously generate novel levels, pushing agents to learn robust strategies that extend beyond memorization. This project aims to develop an RL agent that can effectively master a subset of the games within the Procgen Benchmark, specifically CoinRun, Ninja, and CaveFlyer. The agent is trained using Proximal Policy Optimization (PPO) [3], to optimize performance across these procedurally generated games. The architecture of the models are inspired by IMPALA [2], a highly efficient RL framework designed for large-scale distributed training. This report will describe the two PPO-based agents implemented, their differences, pros and cons, and the evaluation of the most promising one. The decision to implement two different versions comes form the fact that a shared network with two heads is the better choice for efficiency and simplicity in most cases, but separate networks might be more appropriate if task-specific learning and flexibility are critical.

2 Methodology

2.1 Environment Setup

This project uses a subset of environments from the Procgen Benchmark. Specifically, the selected environments are *CoinRun, Ninja,* and *CaveFlyer*, offering distinct challenges and forcing the agent to develop adaptable strategies.

Coinrun A simple platformer where the player starts on the far left and must reach a coin at the far right. The environment presents obstacles like stationary saws, moving enemies, and chasms that lead to death if the player falls.

Ninja Another platformer where the player navigates narrow ledges while avoiding randomly placed bomb obstacles. The agent can throw stars to clear bombs and perform chargeable jumps to reach distant platforms. The goal is to collect a mushroom at the end of the level.

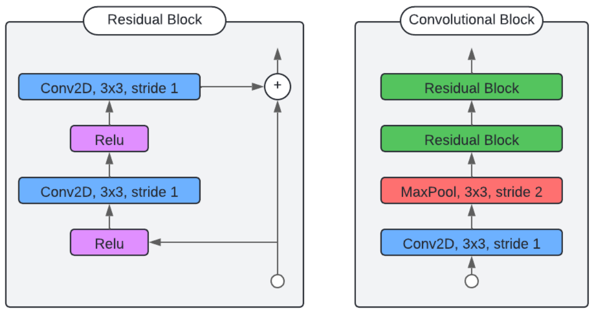


Figure 1: The architecture of the Residual and Convolutional blocks

Caveflyer In this game, the player controls a spaceship navigating through a cave system, similar to the movement mechanics in the classic Atari game Asteroids. The player can rotate the ship and move forward or backward, avoiding obstacles and collecting rewards by destroying targets. The primary objective is to reach a friendly ship at the end of the cave.

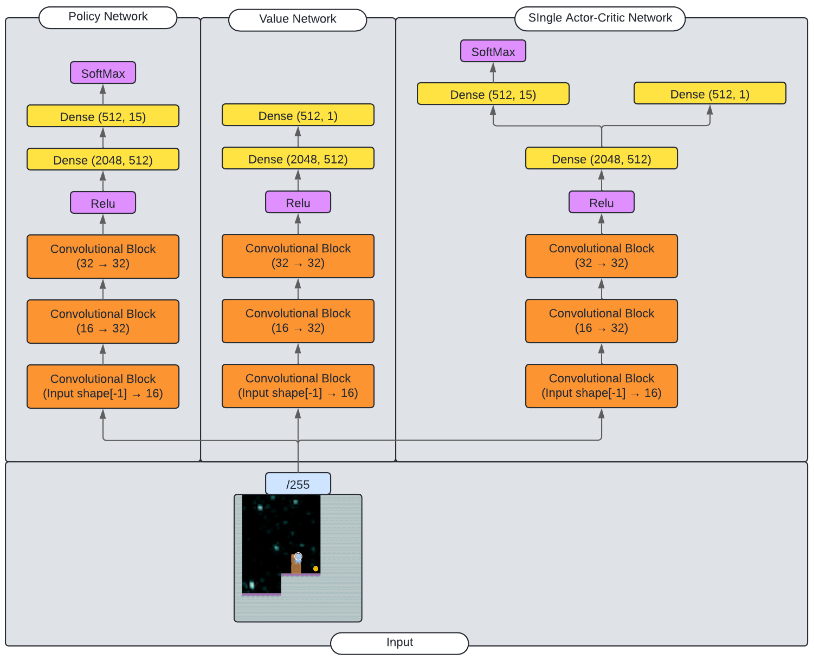
The procgen environments falls into two main categories. In the first one, the objective is to complete a task in the minimum number of time steps to maximize the total reward. In the second, the goal is to avoid events that cause the end of the episode, where maximizing the reward implies maximizing the number of time steps per episode. The three environments chosen in this projects, all falls in the first category. Still, their differences require the agent to adapt to their specific content.

2.2 Model Architecture

The models developed for this project are based on a convolutional architecture with residual blocks inspired by the IMPALA (Importance Weighted Actor-Learner Architecture) design [2]. They are tailored for reinforcement learning environments and follow an actor-critic approach. The models implemented are a total of three: a Value Network, a Policy Network, and an integrated Actor-Critic Network. The three networks are used to successfully implement two different versions of the PPO agent. All three models share a similar base structure that consists of multiple Convolutional Blocks, shown in Figure 1, which are responsible for feature extraction from the input state. Residual blocks, following the IMPALA-inspired design, help with deep networks by allowing gradients to flow more easily through the network, thus mitigating the vanishing gradient problem and improving stability during training. The Policy Network and Value Network are built as two distinct architectures with a shared backbone structure based on convolutional and residual blocks, followed by feed-forward layers and specific output heads. The Actor-Critic Network integrates both policy (actor) and value (critic) predictions into a single model. This network reuses the same convolutional and residual blocks for feature extraction, sharing these layers between the actor and critic. However, the heads are distinct. The architectures of the three different networks are shown in Figure 2

2.3 Training Process

As introduced in the **Model Architecture** chapter, the two implementations of the PPO (Proximal Policy Optimization) agent differ fundamentally in their network architectures and optimization approaches, each offering distinct advantages and limitations. The first version of the agent employs separate networks for the policy and value functions, while the second version utilizes a shared network for both, often referred to as an actor-critic architecture. In the first implementation, the agent uses two independent neural networks: one for the policy (actor) and another for the value function (critic). This separation allows the networks to specialize in their respective tasks: the policy network focuses solely on action selection, while the value network learns to predict the expected return for a given state. One advantage of this approach is the potential for improved performance, as each network can fine-tune its parameters specifically for its task. This specialization also enhances stability during training. Empirically, agents with separate networks often exhibit more stable learning dynamics due to the reduced interference between the actor and critic updates. Since the policy and value networks are trained independently, the updates to the value function do not directly impact the



Policy Network
Value Network
SIngle Actor-Critic Network
SoftMax
Dense (512, 15)
Dense (2048, 512)
Relu
Convolutional Block (32 → 32)
Convolutional Block (16 → 32)
Convolutional Block (Input shape[-1] → 16)
Dense (512, 1)
/255
Input

Figure 2: The architecture of the Policy, Value and single Actor-Critic Networks

policy's action distribution, mitigating potential disruptions during training. In contrast, the second implementation combines the policy and value functions into a single actor-critic network. The shared architecture simplifies the model by using a common set of parameters to output both the action probabilities and value estimates. While this reduces the overall number of parameters and computational cost, it introduces a trade-off. The shared network may struggle with optimizing both tasks simultaneously, as the gradients from the policy and value updates may conflict. This can lead to slower convergence or less stable training, as the network must balance learning both the policy and value functions from a single set of weights.

2.3.1 Loss Functions

The loss functions used in both implementations are based on the PPO objective, but there are some differences due to the network architectures. In both cases, the core of the PPO loss is the clipped surrogate objective, which ensures that policy updates remain within a certain trust region by clipping the probability ratio between the old and new policies. This prevents overly large updates that could destabilize learning. Both implementations also incorporate an entropy term to encourage exploration by penalizing overly deterministic policies. In the first implementation, the policy loss is computed as the minimum of the clipped and unclipped surrogate objectives, ensuring that the policy does not deviate too much from the previous policy.

$$L^{CLIP}(\theta) = -\mathbb{E}_t[min(r_t(\theta)\hat{A}_t^{GAE}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{GAE})] \tag{1}$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{2}$$

is the probability ratio between the new and old policies for the action a_t, $\hat{A}_t^{GAE}$is the advantage estimate, representing how much better (or worse) taking action a_t at state s_t is compared to the expected outcome, and ϵ is the clip range hyperparameter that controls the range of allowable changes. Entropy is added to the loss to encourage exploration. Higher entropy means more randomness in the policy, which prevents the agent from prematurely converging to suboptimal deterministic policies:

$$EnthropyLoss = -\sum p(a|s)log(p(a|s)) \tag{3}$$

Table 1: Training Parameters

N° Levels	200
Difficulty	Easy
Background	True
N° Timesteps Training	200000
Learning Rate	5×10^{-5}
Gamma	0,999
Lambda	0.95
Clip Range	0.2
Enthropy Coefficient	0.01
Batch Size	1024
Epochs	4
Normalize Rewords	True

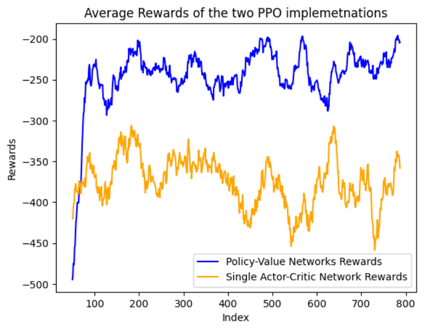


Figure 3: Average Reward of the two PPO implementation during training of the Coinrun game

The total policy loss is the weighted sum of these losses, where entropy is weighted by entropy coefficient hyperparameter c_e:

$$TotalPolicyLoss = L^{CLIP}(\theta) - c_e \times EnthropyLoss \tag{4}$$

The value network's loss is defined as the mean squared error between the predicted value $V(s_t)$ and the actual return R_t which is the sum of rewards the agent expects to receive.

$$ValueLoss = \frac{1}{2}\sum(R_t - V(s_t))^2 \tag{5}$$

This separation of the policy and value losses ensures that each network is updated independently to minimize its respective error.
In the second implementation, the loss function includes a combined policy and value loss, as both tasks are handled by the same network. Here, the value loss is still a mean squared error between the predicted value and the return, but it is weighted along with the policy loss and the entropy regularization:

$$TotalLoss = PolicyLoss + c_v \times ValueLoss - c_e \times EnthropyLoss \tag{6}$$

The fact that the same network is responsible for both tasks introduces potential interference, as the updates for the policy and value functions are entangled.
The table 1 shows the training parameters used during training. The choice has been heavily influenced by the recommended parameters presented in the implementation of the OpenAI researcher [1].

3 Results and Evaluation

The first evaluation performed during the project was to compare the training procedure of the two implementation of the PPO agents. As shown in Figure 3, the first approach, consisting in using two separate networks for the Policy(Actor) and Value(Critic), lead to a more stable training process and a higher average reward. Because of that, the following results refers to this implementation.

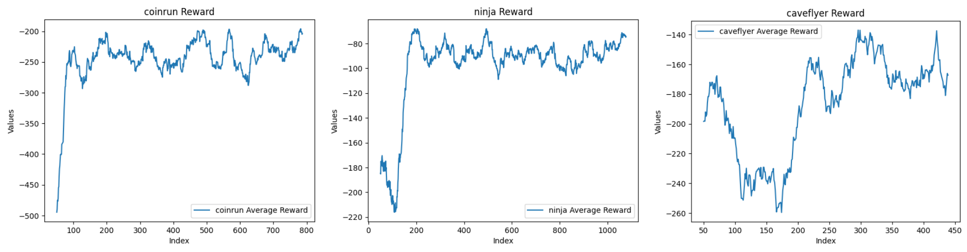


Figure 4: Average Reward of the training process of the Coinrun, Ninja and Caveflyer games, computed with a window of 50 episodes, lasted 200.000 time steps

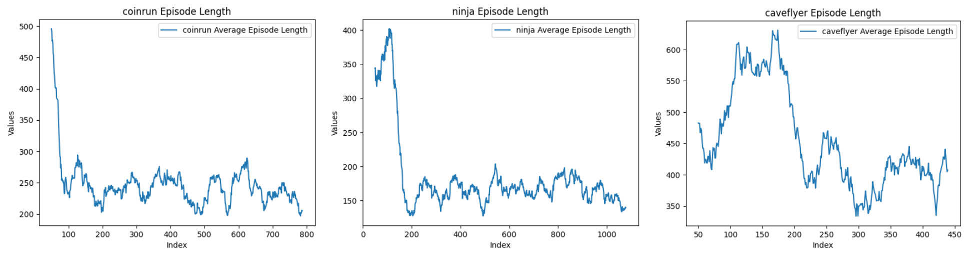


Figure 5: Average Episode Length of the training process of the Coinrun, Ninja and Caveflyer games, computed with a window of 50 episodes, lasted 200.000 time steps

The real evaluation process consists in testing the trained agent for 200 unseen levels, across three different seeds to ensure stability. The same evaluation process is applied to a Random Agent, whose results are used as comparison. The Figures 4 and 5 show the trend of average reward and episode length of the three games, **Coinrun, Ninja** and **Caveflyer**, during the training process. It is possible to see the increase in average reward, and consequent decrease in average episode length, between the beginning and end of training. The Figures 6 and 7 show the trend of average reward and episode length of the three games during the evaluation process, compared with the random agent. The area of the respective color represent the standard deviation computed among three seeds. The table 2 shows the reward and episode length of the PPO and the random agents averaged along the entire evaluation process.

4 Conclusions

In conclusion, this project successfully developed a reinforcement learning agent capable of mastering selected games within the Procgen Benchmark using Proximal Policy Optimization (PPO). The comparative analysis of two implementations, one utilizing separate networks for policy and value functions, and the other employing a shared actor-critic architecture, demonstrated that the former yielded more stable training dynamics and superior performance. It is not possible to exclude

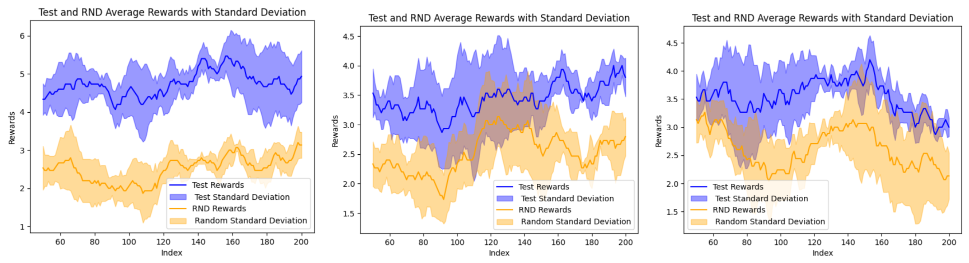


Figure 6: Average Reward of the evaluation process of the Coinrun, Ninja and Caveflyer games, computed with a window of 50 episodes, lasted 200 episodes per seed and compared with the Random Agent

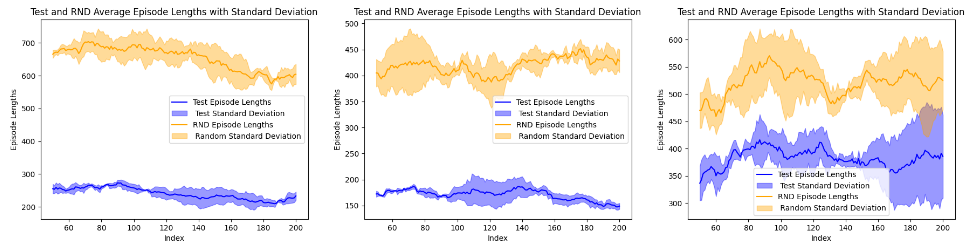


Figure 7: Average Episode Length of the training process of the Coinrun, Ninja and Caveflyer games, computed with a window of 50 episodes, lasted 200 episodes per seed and compared with the Random Agent

Table 2: Evaluation Results

Game	Avg Reward	Avg Ep Lenght	RND Reward	RND Ep Lenght
Coinrun	4,73	244,0	2,58	650,7
Ninja	3,50	167,8	2,49	417,1
Caveflyer	3,50	373,7	2,64	515,3

that this empiric outcome can be the results of implementation errors or an imprecise tuning of the hyperparameters. The trained agent exhibited significant improvements in average reward and reduced episode lengths across the CoinRun, Ninja, and CaveFlyer environments. This emphasizes the importance of architecture choice in reinforcement learning, particularly in complex, procedurally generated settings. A positive impact has been viewed when applying reward normalization during the training process, using the parameters recommended by the researcher of OpenAI [1]. The main limitation of the project has been the computational time needed during the training and evaluation process, limiting the maximum number of training timesteps and the fine tuning of the different parameters of the algorithm. For those reasons, a possible improvements would be increase the training steps and validate different combination of the parameters.

References

References

[1] Karl Cobbe et al. *Leveraging Procedural Generation to Benchmark Reinforcement Learning*. 2020. arXiv: 1912.01588 [cs.LG]. URL: https://arxiv.org/abs/1912.01588.

[2] Lasse Espeholt et al. *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. 2018. arXiv: 1802.01561 [cs.LG]. URL: https://arxiv.org/abs/1802.01561.

[3] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.