Autonomous and Adaptive Systems Project Work with ProcGen Benchmark

Claudia Citera
Master's Degree in Computer Engineering
Alma Mater Studiorum - University of Bologna
claudia.citera@studio.unibo.it

1 Introduction

In this paper, **Proximal Policy Optimization (PPO)** algorithm [1] is used to train an agent to play some **Procgen** games [2]. The **Procgen Benchmark** consists of 16 procedurally generated environments intended to assess sample efficiency and generalization. The tasks of these environments include platforming, navigation, and dynamic control, and each episode is different. Procgen was created to address the **generalization problem** in deep **reinforcement learning (RL)**. Previous work showed that many RL agents tend to overfit with large training sets and fail to adapt to new scenarios. These environments from Procgen emphasize generalization, so that agents perform well in unseen situations. Furthermore, these environments are computationally efficient and can be used for rapid prototyping and large-scale testing.

2 Methods

2.1 Algorithm

As spoiled in the Introduction, the algorithm used for this project is **Proximal Policy Optimization (PPO)**. PPO is a policy gradient method that is popular due to its stability and efficiency. It is suitable for environments with high variability because of its **clipping mechanism** that balances the exploitation and exploration, preventing huge steps in policy.

PPO is based on **Trust Region Policy Optimization (TRPO)**, which is a trust region method that ensures that the new policy does not differ a lot from the old policy. While TRPO relies on second order derivatives, PPO uses a simpler clipped objective function to achieve almost the same stability with less computational load.

PPO uses an **actor-critic architecture**, where the **actor** (*policy network*) selects the actions and the **critic** (*value network*) provides a value function estimate to compute advantages and update the policy. This algorithm uses a probability ratio $r_t(\theta)$ to state how close is the new policy to the old one. The ratio is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{1}$$

where $\pi_\theta(a_t|s_t)$ is the probability of taking action a_t in state s_t under the new policy and $\pi_{\theta old}(a_t|s_t)$ is the probability of taking action a_t in state s_t under the old policy. PPO maximizes the objective function

$$L^{CPI}(\theta) = \mathbb{E}_t[r_t(\theta)A_t] \tag{2}$$

where A_t is the advantage function that estimates how much better (or worse) an action is compared to average action in a given state. To solve the problem of unstable policy updates, the objective function is clipped by a small ϵ:

$$L^{CPI}(\theta) = \mathbb{E}_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{3}$$

Preprint. Under review.

The clipping mechanism enforces the probability ratio to the range $[1-\epsilon, 1+\epsilon]$ to avoid large updates and to maintain stability. Furthermore, PPO combines this clipped objective with a value function loss $L^{\text{VF}}(\theta)$ that minimizes the mean squared error between the predicted value $V_\theta(s_t)$ and the target value (e.g. discounted returns). To encourage exploration, PPO also includes an entropy bonus $S[\pi_\theta](s_t)$. The total loss function is given by:

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t \left[L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{4}$$

where c_1 and c_2 are hyperparameters that are used to control respectively the weight of the value function loss and entropy bonus. This combination helps PPO to achieve robust and stable policy optimization, which makes this algorithm one of the most popular in RL. [1]

2.2 Hyperparameters

In order to train and test the agent, the TensorFlow library was used. One of its main features is `tf.GradientTape` to compute the gradients for the model. Some of the hyperparameters involved include:

- Clipping range (ϵ): This was set to 0.2 as a common value in PPO implementations.
- Discount rate (γ): This was set to 0.99 to put emphasis on long-term rewards and to encourage the agent to prioritize future results.
- Generalized Advantage Estimation (GAE) Lambda (λ): This was set to 0.95 to control the trade-off between bias and variance in the advantage estimates.
- Value function coefficient (c_1): This was set 0.5 to make sure that the value network was trained properly without overriding the policy updates.

Hyperparameter	**Value**
Clipping range (ϵ)	0.2
Discount rate (γ)	0.99
GAE Lambda (λ)	0.95
Value function coefficient (c_1)	0.5

Table 1: Hyperparameters and their values.

In order to improve the training efficiency, both the learning rate and the entropy coefficient (c_2) were tuned using a polynomial decay scheduler. The learning rate scheduler decreases the initial value 5×10^{-4} to a minimum value of 1×10^{-5} over the total training steps. In the same way, the entropy coefficient, that controls the degree of exploration, is reduced from 0.05 to 0.001. This ensures that the agent explores more during the initial stages of training and exploits more as training progresses. The chosen optimizer is Adam, along with the learning rate scheduler.

Training was carried out for about 400.000 steps and updates were made every 4096 steps for 4 epochs. Since 400.000 steps are not sufficient for good performance, the problem of *potential local optima* is present. So in addition to the fully trained model, checkpoints were saved within a 40-episode window whenever the moving average exceeded the previous best one.

2.3 Network Architecture

Due to computational constraints, a single shared network was used for both the actor and the critic to avoid increasing the complexity. The architecture is composed by:

1. **Shared Layers** (for feature extraction):
   - Three `Conv2D` layers with ReLU as activation function, which progressively capture finer details.
   - Each `Conv2D` layer is followed by a `BatchNormalization` layer to stabilize training and a `MaxPooling2D` layer to reduce the input's dimensions.
   - A final `Flatten` layer shapes the extracted features into a 1D vector before passing it through the remaining fully connected layers.

2. **Actor Head**:
    - A `Dense` layer with `n_actions` neurons and a `softmax` activation function, which gives a probability distribution over actions (i.e. the policy).
3. **Critic Head**:
    - A `Dense` layer with a single neuron, outputting the estimated state value. This is used to compute the advantages during training.

During a forward pass, the input is first normalized and then processed by the shared layers. The extracted features are passed to both the actor and critic heads, which generate their respective outputs. These outputs are returned by the network and are used to calculate the policy and value losses.

2.4 Environment Configurations

To optimize the training process given the limited computational resources, the following choices were taken:

- The **difficulty** in all environments was set to '**easy**' (`distribution_mode="easy"`) to make the episodes last less and achieve faster convergence.
- **Black backgrounds** (`use_backgrounds=False`) were chosen to reduce visual complexity.
- The **number of levels** was set at 100 (`num_levels=100`) in training environments, while in testing environments the parameter was set at 200 (`num_levels=200`) to ensure robust generalization.

3 Experiments and Results

The agent was trained and tested in environments with different seeds to ensure generalization. The chosen method to display results was the cumulative reward over 100 episodes, in order to visually understand how the agent behaved in the different games. All the experiments were performed using platformer games with different mechanics and challenges, which is good to test the agent's ability to generalize and adapt across different environments because these games share similar reward distribution. The complexity between the three games increases step by step through CoinRun, Jumper, and then Ninja.

3.1 CoinRun

As described in the Procgen paper [2], Coinrun is a simple platformer, in which the goal is to collect the coin at the far right of the level and the player spawns on the far right. The player must dodge obstacles, enemies, chasms in order not to die.

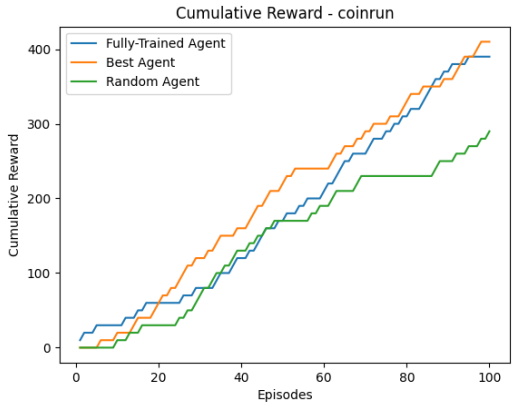


Figure 1: Testing cumulative reward over 100 episodes.

Agent (Eps)	Cum reward	Avg
Full (638)	390.0	3.9
Best (525)	410.0	4.1
Random	290.0	2.9

Figure 2: Coinrun agents testing statistics. *Eps* indicates the episode checkpoint.

3.2 Jumper

As described in the Procgen paper [2], Jumper is a platformer with an open world layout. The player, who is a bunny, must explore the world to locate a carrot. To achieve its goal, the bunny may need to move upward or downward through the level. It also has the ability to double jump, allowing it to navigate tricky layouts and reach high platforms. Throughout the level, there are spike obstacles that will destroy the player on contact. The screen includes a compass, which displays direction and distance from the carrot. The only reward in the game comes from collecting the carrot, at which point the episode ends.

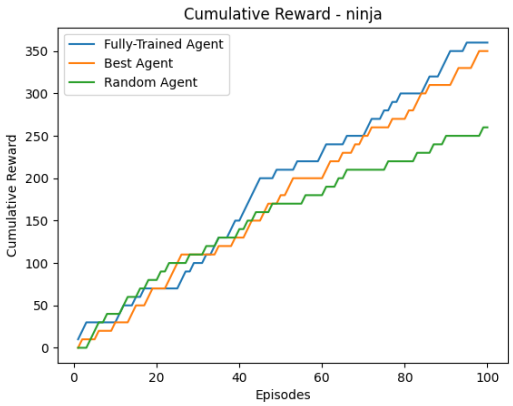
Agent (Eps)	Cum reward	Avg
Full (1134)	360.0	3.6
Best (639)	360.0	3.6
Random	220.0	2.2

Figure 4: Jumper agents testing statistics. *Eps* indicates the episode checkpoint.

Figure 3: Testing cumulative reward over 100 episodes.

3.3 Ninja

As described in the Procgen [2], Ninja is a platformer in which the player, who is the ninja, must jump across narrow ledges while avoiding bomb obstacles. The ninja can throw stars at various angles to destroy bombs when needed. The ninja's jump can be charged over multiple timesteps to achieve greater height or distance. The player earns a reward by collecting a mushroom at the end of the level, which also ends the episode.



Agent (Eps)	Cum reward	Avg
Full (1017)	360.0	3.6
Best (544)	350.0	3.5
Random	260.0	2.6

Figure 6: Ninja agents testing statistics. *Eps* indicates the episode checkpoint.

Figure 5: Testing cumulative reward over 100 episodes.

4 Discussion

The agent was trained on CPU for nearly 400.000 timesteps for each game. This choice was made in regard to both computational and time resources; however, increasing the number of steps would improve the agent's behaviour because it would have the opportunity to learn from new observations.

A key aspect was using schedulers for the learning rate and the entropy coefficient. The learning rate scheduler was used to stabilize training by dynamically adjusting the step size during optimization. As for the entropy coefficient, this was reduced gradually over the training process, which forced the agent to explore the environment more extensively in the early stages and shift towards exploiting learned strategies as training progressed.

It is encouraging that the agent achieves a similar average score across the three games, which suggests that it has generalized well to different game mechanics. Another thing that stands out is the fact that in all three games the agent's performance ceases to improve significantly by 500/600 episodes, which means that the agent has likely identified most basic strategies available in the environment and further improvement can be gained through the use of more advanced techniques or longer training.

As for future developement, residual blocks can be added to handle more feature complexity and LSTM (Long Short-Term Memory) layers for the temporal dependencies by keeping the state that holds the information about the previous time steps. An attempt was made to use a network with residual blocks, but the training (with the same hyperparameters) was too long.

5 Conclusions

As previously said, the main limitation of this project was the computational resources required to get good results. As seen in the Experiments and results section, the agent performs similarly between the different game environments. This suggests that the agent is learning some general behaviors, but may not be fully adapting to the unique challenges of each game. This could be due to different factors like limited training steps, insufficient exploration, and also low adaptability because the same network architecture and hyperparameters were used for all three games.

References

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov "Proximal Policy Optimization Algorithms", arXiv:1707.06347 (2017) doi:10.48550/arXiv.1707.06347

[2] K. Cobbe, C. Hesse, J. Hilton, J. Schulman "Leveraging Procedural Generation to Benchmark Reinforcement Learning", arXiv:1912.01588 (2020) doi:10.48550/arXiv.1912.01588