Autonomous and adaptive systems project report

Guarrera Salvatore

Abstract

This report covers the autonomous and adaptive systems project. The goal is to implement a Reinforcement Learning (RL) algorithm to outperform a random agent in procedurally generated games using the OpenAI Procgen benchmark, focusing on a subset of the 16 Procgen games. The chosen algorithm is Proximal Policy Optimization (PPO). It was implemented and trained on Procgen games CoinRun and FruitBot. In both games, the PPO agent successfully outperformed the random agent.

1 Introduction

The aim of the project is to train an advanced Reinforcement Learning algorithm on the Procgen environment, ensuring it can outperform a random agent in a selection of the 16 Procgen games. Specifically, the training occurs on 200 levels of a game, after which the agent is tested on infinite levels to evaluate its generalization capability. The first major decision involved selecting the appropriate algorithm, which was determined by experimenting with various options. The algorithms tested ranged from the simplest to the most complex, including:

- DQN
- REINFORCE
- A2C
- PPO

Among the algorithms tested, PPO yielded the best results, so it was chosen for further testing. The following sections describe the settings, methods, and hyperparameter tuning. Afterward, the results are presented and final conclusions are stated.

2 Methods

The PPO algorithm is used to train the model on the Procgen environment. The Whole PPO Algortihm is implement from scratch, the main idea is to create a class class called PPOagent that is able to manage the training of actor and critic, along with the possiblity to make agent take actions based on the state that receives from the enviroment. During the training, a rollout of *batch_size* time steps is collected, through the agent interacting with the enviroment and taking actions. Then all the collected data is passed to the PPOagent class that manages to process data and use it for training loop. The process is repeated again until *max_time_steps* is reached.

2.1 PPO Agent

The PPO agent is implemented through a class named *PPOagent*, which consists of various methods:

- **action**: This method allows the actor network to sample an action based on a given state by following the learned policy.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

- **critic_loss**: Computes the critic loss using the mean squared error between discounted rewards and values predicted from the critic network.
- **actor_loss**: Computes the actor loss by implementing the PPO loss, incorporating entropy to increase exploration in the initial episodes. The PPO actor loss can be expressed as:

$$L_t^{(\text{CLIP}+V_F+S)}(\theta) = \hat{E}_t[L_t^{(\text{CLIP})}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \quad (1)$$

- **compute_discounted_rewards** and **compute_advantages**: These methods compute discounted rewards and advantages following the general advantages estimation approach.
- **normalize_rewards**: Adjusts rewards to place them on a common scale.
- **train**: Manages the entire training process, converting and adapting lists into numpy arrays with correct shapes, computing advantages and discounted rewards and handling training for both the actor and critic.

2.2 Neural Network architecture

Two different networks for the actor and critic are used as attributes of the PPO agent. Both networks share a common base network called *BaseModel*. The *BaseModel* network handles processing image patterns and is primarily composed of convolutional layers. This decision makes training faster and leads to better results compared to using two completely different networks for the actor and critic, making it more efficient. Various network architectures were tested, and the final chosen architecture is as follows:

- **Rescaling layer**: Scales the values in the Procgen state from 0-255 to a range between 0 and 1 to prepare them for the convolutional layers.
- **Convolutional layer**: 7x7 kernel size, 16 filters.
- **Convolutional layer**: 5x5 kernel size, 16 filters.
- **Convolutional layer**: 3x3 kernel size, 32 filters.
- **Convolutional layer**: 3x3 kernel size, 64 filters.
- **Flatten layer**: Prepares the extracted features for the dense layer.

Next, distinct classification heads are added on top of the *BaseModel* network for both the actor and critic. Each head consists of a dense layer with a size of 64. In the case of the actor network, an additional dense layer with dimensions corresponding to the action size serves as the output. For the critic network, a dense layer with a single dimension is used as the output.

2.3 Training Main Loop

The primary loop operates over time steps. Initially, a rollout of a specified batch size is performed to collect data for training the agent. The collected data during each rollout includes:

- **actions**: A list containing actions taken during rollout for each timestep.
- **states**: A list containing states obtained during rollout for each timestep.
- **rewards**: A list containing rewards obtained during rollout for each timestep.
- **dones**: A list indicating if the episode ended during rollout for each timestep.
- **probs**: A list containing probability distributions over actions for each timestep during rollout.
- **values**: A list containing values for each state during rollout.

After collecting all this data, it is passed to the agent's train method for model training. This process repeats until the maximum number of time steps is reached.

2.4 Hyperparameter selection

Various experiments were conducted to optimize hyperparameters. The tested hyperparameters included:

- Network architecture (layers, kernel sizes, dimensions)
- Learning rates for the actor and critic
- Rollout batch size
- Gamma
- Actor loss clip parameter
- Entropy parameter to encourage exploration
- Actor loss critic loss parameter
- Training epochs

After numerous trials, the following hyperparameters were selected:

- **Network architecture**: as described in the Neural Network architecture subsection
- **Gamma** = 0.99
- **Actor learning rate** = 1e-4
- **Critrc learning rate** = 1e-4
- **Clip param** = 0.2
- **Entropy param** = 0.00001
- **Critic loss param** = 0.5
- **Batch size** = 512
- **Epochs** = 3

Training involved 200,000 timesteps per game, limited to 200 levels during training and unlimited levels during testing.

2.5 Testing

Tests were conducted on an infinite number of levels, unlike the 200 levels used in training. To compare the performance of the PPO agent against a random agent, rewards were collected over 100 episodes using the random agent first, followed by the PPO agent, and the mean rewards for both were then compared.

3 Results

Two games, **CoinRun** and **FruitBot**, were evaluated using the PPO algorithm, and both games showed superior performance over a random agent.

3.1 CoinRun

CoinRun is a platform game where the agent collects coins while avoiding obstacles. The agent receives a reward of 10 if it collects a coin and completes the level, and 0 if it fails due to traps or enemies. The random agent achieved a mean score of **2.2** over 100 episodes.

The PPO agent was trained for 200,000 timesteps, and Figure1 graph illustrates how the average reward evolved during training.

After training, PPO agent achieved a mean score of **6.7** over 100 episodes, surpassing the random agent by a margin of **4.5**.

The Figure 2 shows episodes scores comparison between random agent and PPO agent in Coinrun game.

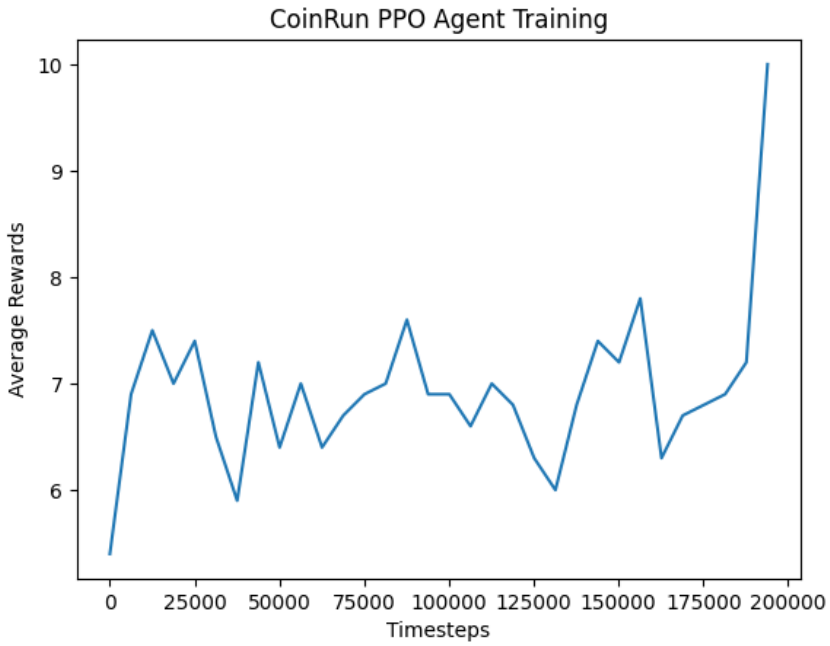

Figure 1: Graph showing Coinrun training progress over timesteps. Collected episode rewards are sliced in window size of 100 and then averaged, in order to show trends during training

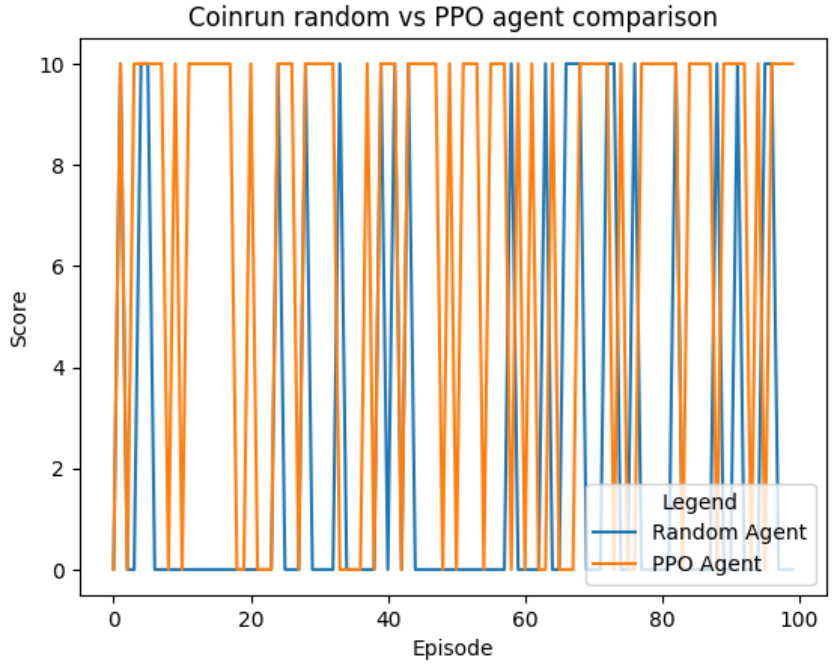

Figure 2: Graph showing comparison between random agent and PPO agent in Coinrun game

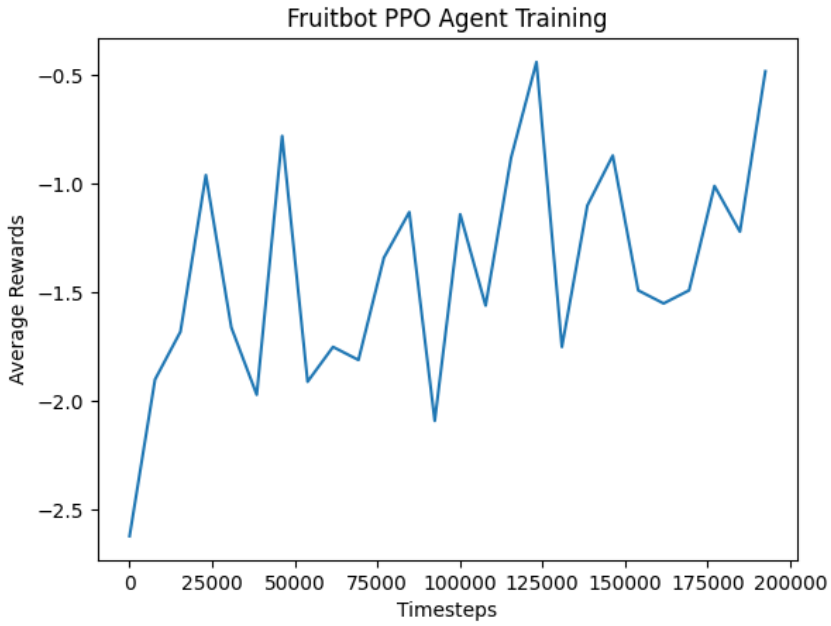

Figure 3: Graph showing Fruitbot training progress over timesteps. Collected episode rewards are sliced in window size of 100 and then averaged, in order to show trends during training

3.2 FruitBot

FruitBot is a navigation game where the agent collects fruits while avoiding non-fruit objects. The random agent achieved a mean score of **-2.65** over 100 episodes.

Similar to CoinRun, Figure 3 illustrates the average reward during the PPO agent's training process.

Post-training, the fruitbot PPO agent achieved a mean score of **-1.39** over 100 episodes, outperforming the random agent by **1.26** points.

The Figure 4 shows episodes rewards comparison between random agent and PPO agent in Fruitbot game.

4 Conclusions

In conclusion, the PPO agent demonstrates superior performance compared to a random agent in two games successfully. Future experiments could be performed by testing additional games from the set of 16 available in Procgen, or by exploring alternative hyperparameters to further enhance results and increase the average score.

5 References

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. Proximal Policy Optimization Algorithms: https://arxiv.org/abs/1707.06347

Karl Cobbe, Christopher Hesse, Jacob Hilton, John Schulman. Leveraging Procedural Generation to Benchmark Reinforcement Learning: https://arxiv.org/abs/1912.01588

OpenAI procgen GitHub: https://github.com/openai/procgen

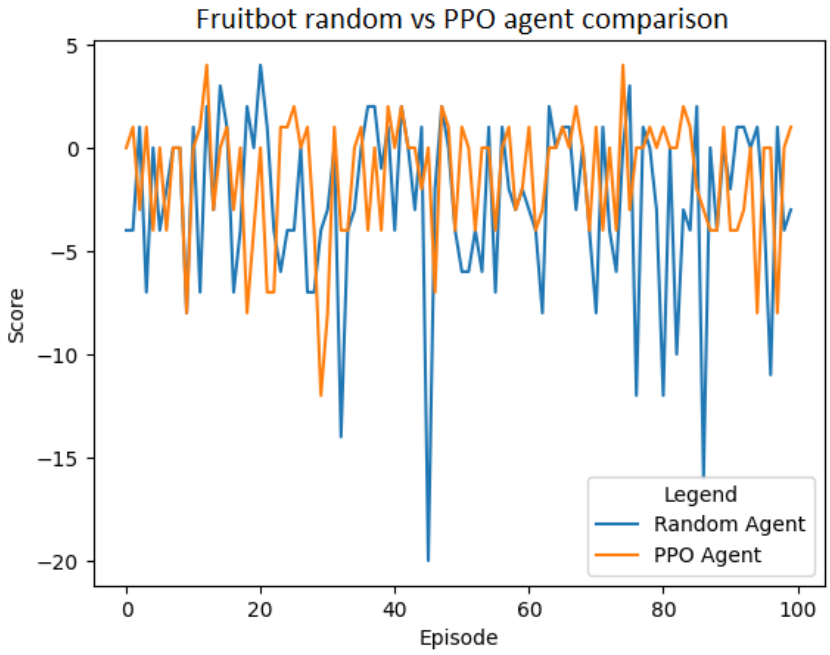

Figure 4: Graph showing comparison between random agent and PPO agent in Fruitbot game