Reinforce benchmarks in Procgen Environments

Luan Cao Vo Tran
University of Bologna
Autonomous and Adaptive Systems
luancaocaovo.tran@studio.unibo.it

Abstract

This paper will explore our implementation of a reinforcement learning algorithm applied to two games within the ProcGen OpenAI Gym environment, aiming to develop an agent that learns to play these games on its own. The reinforcement learning agent is trained to play these games using RGB frames as input, achieved through a combination of a convolutional neural network and the REINFORCE algorithm with baseline. We will highlight our implementation choices and lastly evaluate the results of our implementation.

1 Introduction

The 16 environments provided by OpenAI's Procgen framework are procedurally generated atari-style games with varying play styles. [1]. The reinforcement learning agent used for the problem will be based on a convolutional neural networks combined with the REINFORCE algorithm with baseline [7].

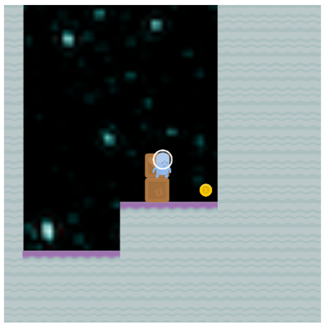
2 Formatting the environments

The procedural nature of the environments are intended to serve as an apt training ground to train more general reinforcement learning agents in as opposed to "fixed-level-style" games. For this paper, the environments used are:

1. **Coinrun**: A simple platformer where the player starts on the far left and must reach the coin on the far right. The game end when the player comes in contact with stationary saws, pacing enemies or falls down deadly chasms.

2. **Leaper**: The player crosses lanes to reach the finish line and earn a reward. First, avoid cars; then, hop across logs on a river. Falling in the river or being run over ends the game.

A screenshot of each of the respective game environments are shown in figure 1.


(a) Coinrun environment [6]

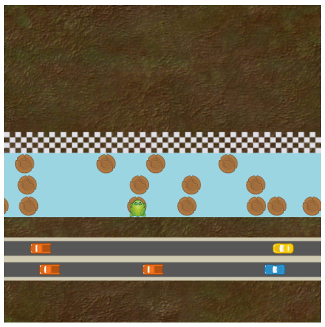

(b) Leaper environment [6]

Figure 1: Chosen environments: Coinrun and Leaper.

From the output of our environments we get our game states, which is a 64×64 RGB image. Our agent can perform actions given the states, which are defined in related code base from Procgen[6]. For the Procgen environment the 15 actions are standardised, meaning that all actions does not necessarily correspond to an input that affects the environment for each game.

2.1 Framestacking

Both of our games contain moving objects which necessitates the use of framestacking such that the direction of objects can be inferred from consecutive frames. In dynamic environments where the agent has to move in relation to another moving object, 1 frame will simply not provide enough information about the game state. Our state is therefore a stack of 4 64×64 RGB frames where the initial state's 4 frames are entirely comprised by the first frame in a brand new episode.

2.2 Convolutional neural networks

Since our state space, S, is large, $S = (256^3)^{4\times 64\times 64}$, we will use convolutional neural networks to extract the most important features of each image to reduce our state space into a more manageable size. With a reduced state space it will be faster to train a reinforcement learning agent, such as one we will use with the REINFORCE with baseline algorithm. Our implementation of this convolutional neural network is inspired from the paper *Playing Atari with Deep Reinforcement Learning*[2].

3 REINFORCE algorithm with baseline

REINFORCE is a Monte Carlo Policy Gradient control algorithm. It is based on estimating the policy function which is the probability distribution of an action given state. It is given as:

$$\pi(a_t \mid s_t; \theta) \tag{1}$$

Where θ is a parameter vector of the policy function. The main goal of a policy gradient method is to learn the policy parameter θ with respect to a performance measure based on $J(\theta)$. If we define the performance measure to be the true value of our policy function $J(\theta) = V_{\pi_\theta}$, where π_θ is the policy determined by θ we get the following update formula for the REINFORCE algorithm by applying the Policy Gradient Theorem [7]:

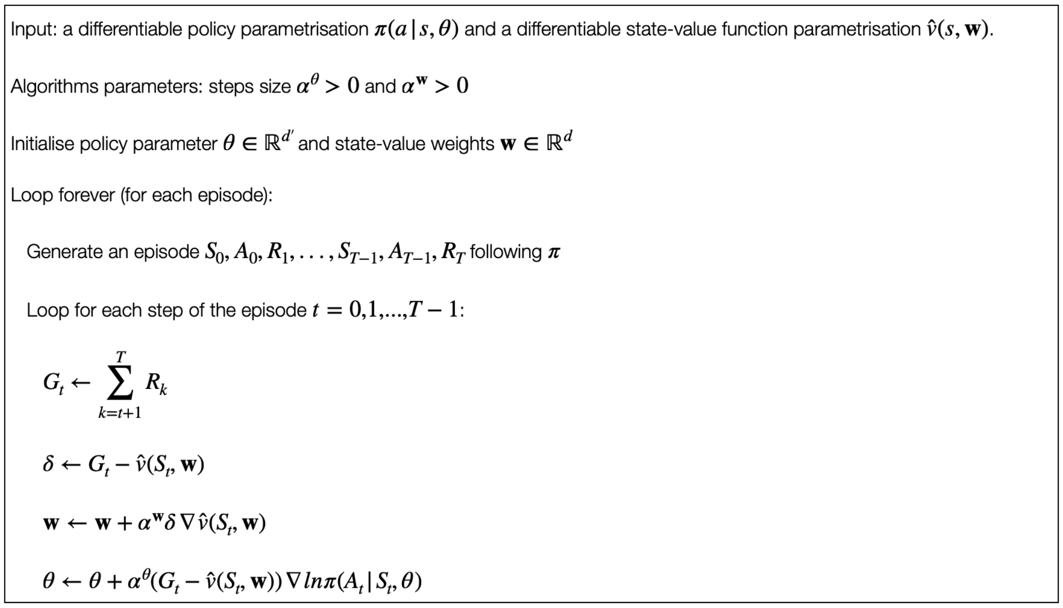
$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \log \pi_\theta(A_t|S_t) \tag{2}$$

REINFORCE being a Monte Carlo method, makes sure it is unbiased because it calculates the return G_t exactly from the complete sequence of rewards obtained during an episode. However, the method suffers from high variance since the cumulative reward G_t can vary significantly between episodes. The method can also be quite slow since it always need to wait for a full episode to finish before applying updates.

One way to combat variance is to introduce a baseline, $b(s_t)$ to our update rule. A natural choice is to learn the state value at the same time $b(s_t) = V(s_t, w)$, where w is parameter vector for the state value function. Since $V(s_t, w)$ is an estimate of the expected return from state s_t subtracting $V(s_t, w)$ from G_t centers the reward around the expected return, which reduces the variance of the gradient estimates. The resulting policy gradient then becomes:

$$\theta_{t+1} = \theta_t + \alpha(G_t - V(s_t, w))\nabla_\theta \log \pi_\theta(a_t|s_t) \tag{3}$$

We can now define the full REINFORCE algorithm with baseline as follows:



```
Input: a differentiable policy parametrisation $\pi(a \mid s, \theta)$ and a differentiable state-value function parametrisation $\hat{v}(s, \mathbf{w})$.

Algorithms parameters: steps size $\alpha^\theta > 0$ and $\alpha^\mathbf{w} > 0$

Initialise policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$

Loop forever (for each episode):

  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi$

  Loop for each step of the episode $t = 0,1,...,T-1$:

    $G_t \leftarrow \sum_{k=t+1}^{T} R_k$

    $\delta \leftarrow G_t - \hat{v}(S_t, \mathbf{w})$

    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^\mathbf{w} \delta \nabla \hat{v}(S_t, \mathbf{w})$

    $\theta \leftarrow \theta + \alpha^\theta (G_t - \hat{v}(S_t, \mathbf{w})) \nabla ln\pi(A_t \mid S_t, \theta)$
```

Figure 2: REINFORCE with baseline from lecture [3]

To compute the weights, w, for the state value estimate, $V(s_t, w)$, the cost function $J(w)$ is defined as:

$$J(w) = (G_t - V(s_t, w))^2 \tag{4}$$

Since G_t is our target for the state value estimation [4].

To implement the algorithm we will use neural networks to learn the values of θ and w that minimises the cost functions of each of the respective parameters. The parameters θ will be the weights of the policy network which will output an action probability distribution which will be sampled to get an action to be taken given the states. w will be the weights for the state value network which outputs the expected return given the state. The implementation in code adapts the Actor-Critic framework provided by Keras[5] in conjunction with a convolutional neural network architecture inspired by the paper *Playing Atari with Deep Reinforcement Learning* [2] to implement a REINFORCE algorithm with baseline that works with RGB images.

4 Implementation Details

The initialisation of the network weights θ and w is the default one provided by Keras. The network features a shared convolutional architecture base followed by two branches: a policy network and a value network. The base includes three 3D convolutional layers with ReLU activation. The value network connects to this network through 2 fully connected layers and has a single output. To define our policy network's output $\pi(a_t \mid s_t; \theta)$ we first needed to reduce the action space. As mentioned earlier, not all of the 15 inputs provided by Procgen mapped to a corresponding in-game action that could affect the environment. By reducing the action space we hopefully increased the training stability since we reduce the effect of multiple inputs bloating our action space.

With the action space defined, we parameterise the output of our policy network $\pi(a_t \mid s_t; \theta)$ as a softmax distribution vector, where the dimension of the output vector matches the size of the action space. This parametrisation gives us a probability distribution over the action space. We then sample an action from this distribution, execute the action, observe the outcome, and use the REINFORCE algorithm with a baseline to update our policy parameters θ in a way that maximises the expected reward. This parametrisation called is softmax in action preferences. The stochastic nature of softmax in action preferences facilitates a balance between exploration and exploitation, removing the need for an exploration parameter, ϵ, which can be found in value-based methods. This is crucial in environments where exploration is necessary to discover optimal strategies. Another advantage of using softmax over action preferences is that it allows the approximate policy to converge towards a deterministic policy if needed. Unlike specific action values, action preferences are adjusted to yield the optimal stochastic policy. If the optimal policy happens to be deterministic, the preferences for the optimal actions will increase significantly relative to sub-optimal actions. As a result, the output of the softmax will approach 1 for the optimal actions, effectively leading to near-deterministic behaviour.

To make training easier I used the ProcGen environment's configurations to make the state space more streamlined for reinforcement learning. This included getting a level distribution of easier levels, reducing the amount of levels to 200 and reducing the environment's colourful and varied sprites to simple cubes that the networks could more easily interpret [1].

The hyperparameters such as learning rate and dropout-rate were arbitarily chosen.

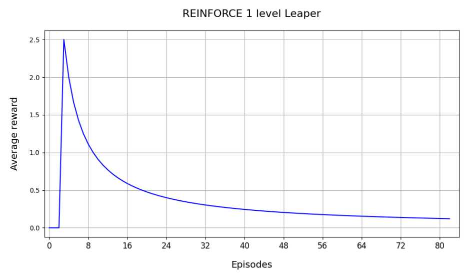
Since the chosen games' reward structures were sparse, only giving a reward at the end, and the step horizon of both games had a maximum count I chose $\gamma = 1$. A high γ may also help long term planning since the rewards are not discounted.

5 Results

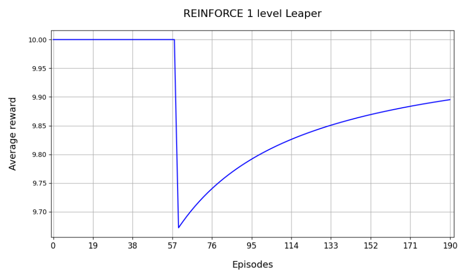
To measure the performance of our models we look at how the average reward changes over the course of our episodes. We will look at two cases specifically:

1. The performance over 1 level
2. The performance over 200 levels

All the training has been done with Procgen's easy configuration.



(a) REINFORCE 1 level Leaper



(b) REINFORCE 1 level Leaper

Figure 3: Performance of REINFORCE on the 1 level Leaper

We start with looking at the Leaper environment. From figure 3 we immediately spot an interesting case. The difference between the two plots is that for figure 3(b) it starts with randomly getting a reward on the first try, whereas for figure 3(a) the agent manages to get one reward after a while, but struggles to reap any others for the rest of the duration under training. As observed under training for the model in figure 3(a) quickly got stuck, refusing to move and frequently using the max steps of

500 for each episode. Training for the model in figure 3(b) on the other hand went quickly. Obtaining a reward at the start of the episode seemed to immediately nudge the action probability distribution in the right direction. As observed around episode 60 the agent missed a reward, but quickly returned to reaping rewards where it was observed that the agent thereafter only used 12 steps, seemingly nearing a deterministic policy. Both models were trained with a number of timesteps far below the one recommended from OpenAI [1], which for the easy configuration is 25 million steps. This harshly affects our ability to correctly evaluate our models, but we still have things to point out. For the model in figure 3(a) if given enough timesteps would perhaps theoretically converge given its good convergence properties, but due to constraints in both time and compute power there are no results exploring this further for a single level case. It is still interesting that we can obtain a result such as one for the model in figure 3(b) for a very low number of timesteps.

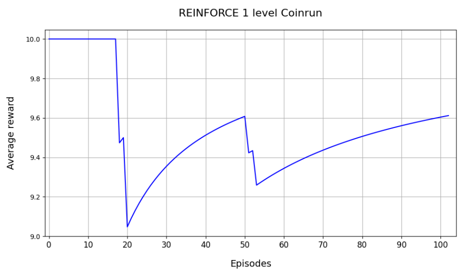
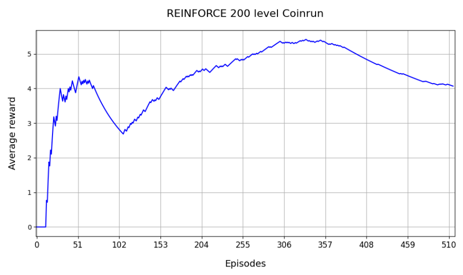


Figure 4: REINFORCE 1 level Coinrun

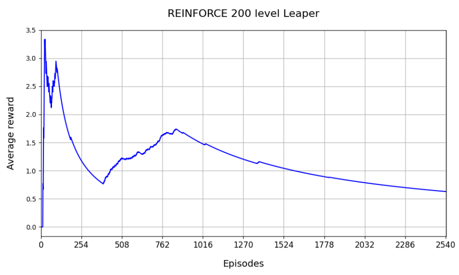
The same situation was also observed for the coinrun environment as seen in figure 4. It seems that in cases where the agent obtains a reward in the first episode, it quickly learns the level for the single level case, and we can safely assume that it only knows the level, not necessarily how to play the game.

A possible reason for the sudden drop-off of learning in the cases found in figure 3(a) compared to figure 3(b) despite a small number of timesteps may be due to how our rewards are separated. Since we only get a reward at the end of an episode, the update step for θ in figure 3(a) may lead to very slow learning due to the rewards being 0 most of the time and the state-value function not being correctly estimated at the start. This in combination with REINFORCE being a Monte Carlo method subjected to slow learning and high variance may leads to slow results. However, obtaining a reward relatively early may perhaps update the values in θ and w just enough to snowball further leading to the quick result in figure 4 and figure 3(b).

We also have not tuned our hyper-parameters such as learning rate to see at how it affects performance, so that may as well be another factor.



(a) REINFORCE 200 level Coinrun



(b) REINFORCE 200 level Leaper

Figure 5: Performance of REINFORCE on 200 levels in the Coinrun and Leaper environments.

In figure 5(a) and figure 5(b) we can see the best results from training the agents on 200 levels. Despite the reward system for both games being the same with a reward of 10 after every successful episode, we can observe that the Coinrun agent has mediocre performance, whereas in Leaper the performance is not adequate and the reward is quite low. This may be due to the Leaper environment being generally more difficult than the Coinrun environment.

We also observe that the training seems unstable, suddenly dropping off before climbing up. This may simply be the combination of the high variability of the levels, high variability of Monte Carlo methods and the complexity of the environments leading to unstable learning. Despite this, at least for the case of figure 5(a) there seems to be an upwards trend if it were to follow the dip-and-climb tendency.

It is important to restate that since we are not close to approaching the Procgen's recommendation of 25 million timesteps, where for our case we have approximately less than 1 million steps the model will not perform well. Especially when considering that the OpenAI also used PPO, an algorithm which learns better than regular REINFORCE, we perhaps would need a timestep threshold higher than 25 million.

Another reason for the poor performance was perhaps that not enough work was done to format the state space into a more manageable size. Despite using a convolutional neural network to reduce the state space, we could use more techniques to further reduce the state space, making training more stable and faster. Pixel normalisation, reducing the range of the pixels from $[0, 255]$ to $[0, 1]$ would improve performance. This in combination with turning the image from RGB to grayscale would greatly reduce the state space going from 3 channels of $0 - 255$ to only 1.

Due to how the REINFORCE algorithm is structured we would perhaps have more success with another game that has a less sparse reward structure.

6 Conclusions

This report has documented the implementation of REINFORCE with baseline on two of the environments in the OpenAI ProcGen, namely Coinrun and Leaper. The implementation has also utilised framestacking in combination with convolutional neural networks to ensure faster training and better performance in the given environments. However, the results indicate that the algorithm, in its current state, struggles to learn a general policy capable of fully solving a Procgen game, despite showing some potential in the specific case of 1 level. While a combination of enhancements regarding state space formatting, hyperparameter tuning, and especially more training could improve performance, the last point in particular rings true for this project as we needed more data to train the model to further evaluate it more correctly.

References

[1] Karl Cobbe et al. "Leveraging Procedural Generation to Benchmark Reinforcement Learning". In: *arXiv preprint arXiv:1912.01588* (2019).

[2] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602* (2013). URL: `https://arxiv.org/abs/1312.5602`.

[3] Mirco Musolesi. *Policy Gradient Methods*. 2023. URL: `https://www.mircomusolesi.org/courses/AAS23-24/AAS2324_PolicyGradientMethods.pdf`.

[4] Mirco Musolesi. *Value Approximation Methods*. 2023. URL: `https://www.mircomusolesi.org/courses/AAS23-24/AAS23-24_ValueApproximationMethods.pdf`.

[5] Apoorv Nandan. *Actor Critic Method*. 2020. URL: `https://keras.io/examples/rl/actor_critic_cartpole/#implement-actor-critic-network`.

[6] OpenAI. *Procgen Benchmark*. GitHub repository. 2020. URL: `github.com/openai/procgen`.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Chapter 13. The MIT Press, 2018.