Autonomous and Adaptive Systems Project Work

Virginia Nucci

Master's Degree Course in Computer Engineering,
Alma Mater Studiorum Bologna

September 22, 2024

Abstract

The aim of this project work for the 2023/2024 Autonomous and Adaptive Systems Course is to understand and explore generalization in Reinforcement Learning (an agent ability to adapt to new unseen situations using knowledge acquired from prior experiences) through the implementation of an algorithm able to learn generalizable skills. To achieve this, the project makes use of ProcGen, a well known benchmark for generalization, that, through procedurally generated content, assess the generalization capabilities of RL agents in diverse environments.

Contents

1 Introduction

In Reinforcement Learning, the use of deep neural networks as non-linear function approximators is very common and encouraged, as researchers has obtained significant success trough their implementation, solving problems in high dimensional states or action spaces. While the advantages of the use of artificial neural network are widely known, issues like overfitting limit the robustness and generalization capability of an agent. Generalization in Reinforcement Learning is a quite recent concern, defined as an agent capability to adapt to unseen situations given the knowledge acquired from prior experiences. A policy, for a certain RL agent, that is learned and evaluated in the same environment can indeed lead to overfitting: the said agent is just learning a fixed sequence of actions that won't be useful in another environment.

To address this issue, a widely common benchmark to evaluate generalization is used in this project, ProcGen[1]: through procedurally generated content, ensures that different versions of the same environment (levels/episodes) are generated: in this way, the agent will learn to play from a selection of levels, an then be able to perform well even in unseen levels. To study generalization, for every game two environments were used and initializated with the same seed, to ensure that both environments started from the same point to generate levels. Due to limited compute resources, the distribution of the levels was set to 'easy' to reduce the number of timesteps required to solve the game, and for the same reason the game used black backgrounds, as it reduces complexity. While for the training environment the number of unique levels that can be generated was 200, for the test this paramether was set to use unlimited levels.

The methods implemented, the results obtained and some further considerations, as well as conclusions, will be found in the next sections.

2 Methods

The algorithm chosen to be implemented in the project work is a variant of A2C, Advantage Actor Critic (Figure 1), an on-policy gradient method from the actor-critic 'family' of algorithms derived by REINFORCE with the addition of a baseline b_t. Actor critic methods learn approximations to both policy and value functions, to reduce variance and accelerate learning; the value function (approximated by the critic network) in this kind of methods is used to assess the quality of a certain action given by the policy distribution (output of the actor network). In A2C, the critic role is taken by the advantage function, used to calculate how better is to select a certain action in a certain state, given the average value of the state. The advantage function used to scale the policy gradient is estimated by the quantity:

$$A(a_t, s_t) = Q(a_t, s_t) - V(s_t). \tag{1}$$

In this scenario, R_t (the return) is typically used to estimate $Q^{\pi}(a_t, s_t)$ (defined as the action-value function), while $V^{\pi}(s_t)$ (the value function) is used as b_t (the baseline).

In A2C, the agent maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$: the policy and the value function are updated whenever an episode ends. The update performed can be seen as :

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta')(R_t - V(s_t;\theta_v)) \tag{2}$$

The agent implements GAE (Generalized Advantage Estimator)[4] to compute the return R_t; advantage is computed with the exponentially-weighted average of n-step returns with decay parameter λ as follows:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} := (1-\lambda)(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \dots) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V, \tag{3}$$

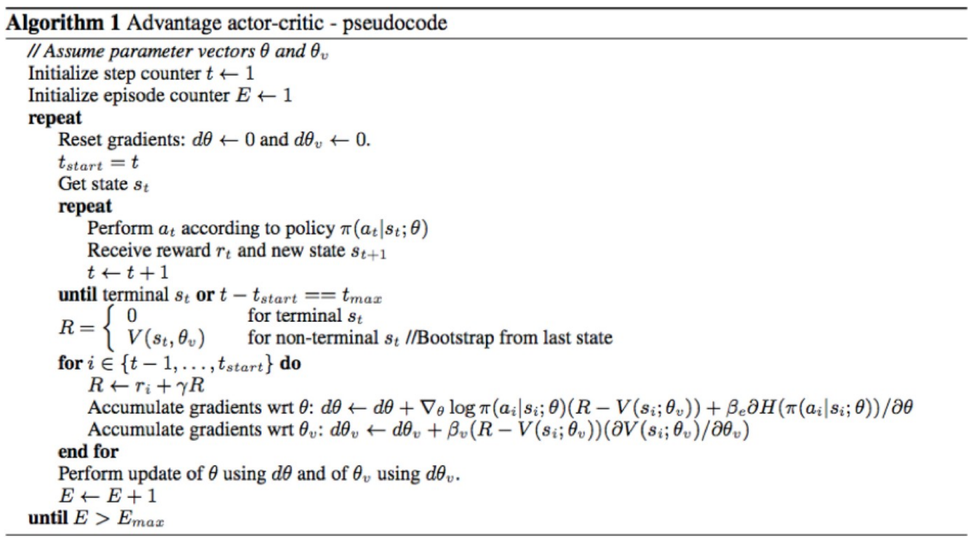
where δ_t is the temporal difference residual. This technique is particularly useful to trade off between bias and variance of the estimator through tuning of the λ paramether: a value close to 1 reduces bias and increases variance.

The network architecture implemented takes inspiration from the IMPALA[2] network, an highly efficient implementation of A3C (Asynchronous Advantage Actor Critic[3]). Experimenting with a model size of this kind helped with the complessity of the observation space. In the convolutional network implemented, the policy network and the value network shares all the layers but has different heads, one for the policy, with a softmax output, and one for the critic, with a linear output.

To improve exploration avoiding premature convergence, the entropy of the policy was also added to the full objective function as implemented in A3C, that becomes:

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta')(R_t - V(s_t;\theta_v)) + \beta \nabla_{\theta'} H(\pi(s_t;\theta')), \tag{4}$$

where β is the coefficient of the entropy regularization.



Algorithm 1 Advantage actor-critic - pseudocode

```
// Assume parameter vectors $\theta$ and $\theta_v$
Initialize step counter $t \leftarrow 1$
Initialize episode counter $E \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta)$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta_v) & \text{for non-terminal } s_t \text{ //Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a_i|s_i;\theta)(R - V(s_i;\theta_v)) + \beta_e \partial H(\pi(a_i|s_i;\theta))/\partial\theta$
        Accumulate gradients wrt $\theta_v$: $d\theta_v \leftarrow d\theta_v + \beta_v(R - V(s_i;\theta_v))(\partial V(s_i;\theta_v)/\partial\theta_v)$
    end for
    Perform update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
    $E \leftarrow E + 1$
until $E > E_{max}$
```

Figure 1: Pseudocode for Advantage Actor Critic.

2.1 Implementation Details

In this section some of the implementation details can be found. The project make use of the tensorflow library to train and test the agent: in particular, a tf.GradientTape object is implemented to compute the gradients for the model. The γ and λ paramethers used to compute the rewards using GAE were set to 0.9 and 0.95 respectively, as common practice. Regarding the advantage used to scale the policy loss, standardization was used to ensure stability through the update process of the network. The value for the entropy coefficient was set to 0.05 after some tuning steps. The total loss of the network is computed as the sum of the actor loss and the critic loss terms, where the critic loss was discounted by a coefficient of 0.5. For the network architecture, an IMPALA variant (without the LSTM layer) was used (see A), with the addition of batch normalization layers in the convolutional blocks (the base module of the structure) to stabilize training. The optimizer chosen in this implementation is the non-centered RMSProp with an initial learning rate of $7 \times e^{-6}$, with a PolynomialDecay scheduler.

3 Results

The agent performance was tested on a ProcGen environment where the number of unique levels generated was set to unlimited as to assess the generalization capability of the agent. To show the results (Figure 2, Figure 3) the cumulative reward over timesteps was chosen, as to visually assess the behaviour of the agent through training.

3.1 Coinrun

As reported in ProcGen documentation, Coinrun is a simple platformer where the goal is to collect the coin at the far right of the level, while the player spawns on the far left. The agent must also dodge obstacles and enemies. The player obtain a reward of 10 when collecting the coin and 0 for all the other timesteps in the episode. In Figure 2 it can be seen how the cumulative reward increase linearly with the number of timesteps. This shows that the agent is quite able to generalize over the levels. The learning rate was set to $7 \times e^{-6}$: increasing this value led to the player repeatedly perform the same action, resulting in convergence to a local minimum.

3.2 Maze

As reported in ProcGen documentation, the player must navigate a maze to find the sole piece of cheese and earn a reward. Mazes are generated by Kruskal's algorithm and range in size from 3x3 to 25x25. The player may move up, down, left or right to navigate the maze, and earns a reward of 10 when collecting

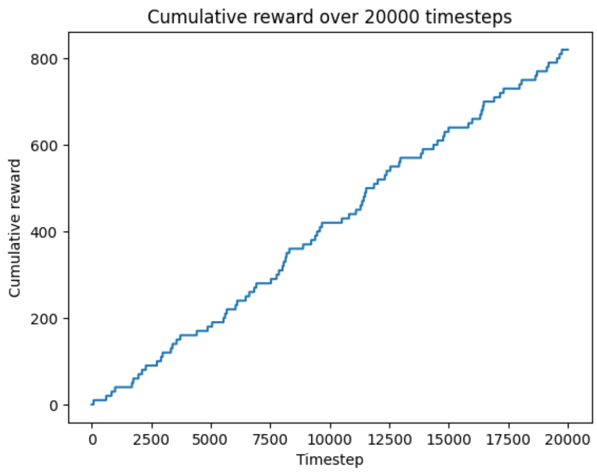


Figure 2: Cumulative rewards over total timesteps on Coinrun.

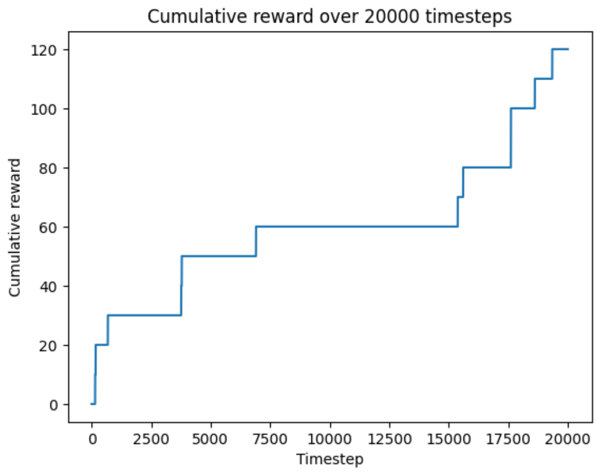


Figure 3: Cumulative rewards over total timesteps on Maze.

the cheese but nothing else for all the other timesteps in the episode. As close as it may seem to the reward function design of Coinrun, the key in this environment was to maximise the exploration to find the coin trough the levels: while in the previous game the coin was always in the same spot, in this case the agent has to find the correct path, that is always different in lenght and complexity. The added complexity from the maze generation pose a different challenge, and in this case the agent, which was trained with the same paramethers as before, did not perform well, as can be seen in Figure 3.

4 Discussion

The agent trained on CPU for 20'000 timesteps for every game. The total number of timesteps is low due to the limited computational resources and time constraints. Increasing the number of steps will be beneficial as the agent will learn to adapt to new observations more accurately.

During the training, the crucial step was tuning the learning rate of the network, as well as the entropy coefficient. When training for a small number of timesteps, increasing the entropy often led to better results, as it enhances exploration. Increasing the learning rate often led to poor performance because of the policy becoming too deterministic.

The chosen model architecture (Impala) could be revised to add an LSTM layer to better deal with the temporal correlations of the different timesteps of an episode. Furthermore, it can be interesting studying how varying the model architecture can impact both on training time and performance, as it's common for larger size networks to exhibit reward hacking.

An approach that would be interesting to implement is that of the frame of reference, a technique adopted typically in in settings in which an agent has to generalize across tasks/environments, where the key idea is to simplify the state spaces defining a new coordinate system. If for example, a frame of reference was implemented in the maze environment, one could think of the direction that the player is facing in the maze as one of the axis in a new coordinate system that observations and actions depend on. For example, now the actions could include going forward or backwards or rotate, simplifying the choice. In this way, the agent now does not need to learn multiple different mappings of observations to actions.

5 Conclusions

To study the generalization capability of an RL agent, a simple A2C algorithm was implemented. As seen in the results section, the agent performs very differently playing the two games, and this is mostly due to the differences between environments: while in Coinrun the agent learns pretty fast which are the best and worst actions with the advantage help(and termination of an episode due to the agent touching an enemy), in Maze the challenge is to maximise the exploration of the labyrinth, and an episode often terminates due to time constraints. One could expect the agent to behave similarly in both, considering how similar the reward functions are designed, but the diverse set of skills required pose an additional challenge.

References

[1] Leveraging Procedural Generation to Benchmark Reinforcement Learning
`https://arxiv.org/abs/1912.015881`

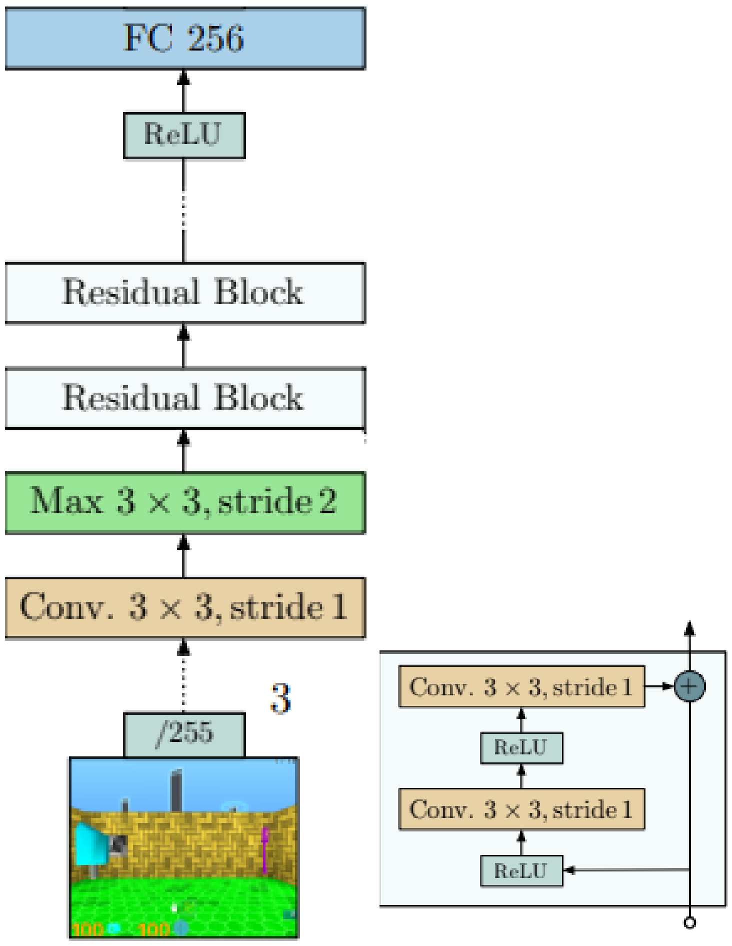
[2] IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures
`https://arxiv.org/abs/1802.01561`

[3] Asynchronous Methods for Deep Reinforcement Learning
`https://arxiv.org/abs/1602.01783`

[4] High-Dimensional Continuous Control Using Generalized Advantage Estimation
`https://arxiv.org/abs/1506.02438`

A Appendix: Model Architecture

The IMPALA convolutional architecture following is used to provide an idea of the model implemented in this report: in each residual block, an additional layer of batch normalization is included. After the final Dense layer, two outputs are defined as the actor and the critic output: a dense layer for the critic and a dense for the actor to compute the logits; applying the softmax to the logits is done later while updating the network.



(a) Convolutional network (b) Residual Block

Figure 4: Impala Architecture