Project Autonomous and Adaptive Systems 2021-22

Giovanni Minelli
`giovanni.minelli2@studio.unibo.it`

Abstract

This project aims to study procedural content generation (PCG) by taking into account two aspects of a game: the player and the game world. The goals were to create an agent (called Generator) capable of generating challenging but not impossible scenes that can be presented to a second agent (called Solver), who by facing new situations learns through experience. The implementation of this pipeline results in a tunable game environment that can create scenes based on a parametrized difficulty value, and an autonomous player agent that achieves better generalizability and overall higher game scores.

1 Introduction

Reinforcement learning has long been involved in the video game industry due to its human-like or even superhuman performance in some tasks. Usually applied in-game, AI-controlled opponents and unscripted behaviors attributed to game elements can make the experience more fun and challenging, but additional uses can also be found during the development phase. One challenge in game development is that environments, characters, resources, etc. can change often, especially during development cycles, and RL find its spot here by allowing the creation of agents with more flexibility and generalizability, and so making continuous re-training almost unnecessary in order to test the updated environment. On the other hand, reinforcement learning can also be very useful in generating new environments, not just playing with them. Instead of exploring the space of possible combinations of assets, one can learn a policy related to the ability of players (agents or humans), to dictate the environment generation thus giving space to more enjoyable gameplay. By focusing on state representation and reward signals between a trainer and a learner, both goals can be targeted and achieved hand in hand. Here we let an agent training on generated scenes that change dynamically in terms of difficulty, showing how this makes him more proficient in dealing with different game situations, and we also implement a map generator as agent, that, being aware of the player's performance, can adapt its output based on a parameterized difficulty value.

1.1 Related work

This work is mainly inspired by previous attempts in this field. In particular, (1) attempted to formulate the procedural content generation problem as an RL problem testing different representations to be used as input for an agent in charge of modifying the game space. (2) aims at the testing phase of game development, using a generating agent to place game objects in a 3D space as adversarial actions and focusing on the agent player's ability to overcome difficulties while playing.

2 Framework

Since very customized interactions with the environment are needed to build the desired pipeline, I created an appropriate game environment, using Pygame as graphic library. As a whole, it is a simple 2D vertically scrolling terrain on which the player (a skier) must move left or right with different angles available [-2, -1, 0, 1, 2] to touch rewarding objects (flags) and avoiding damaging obstacles

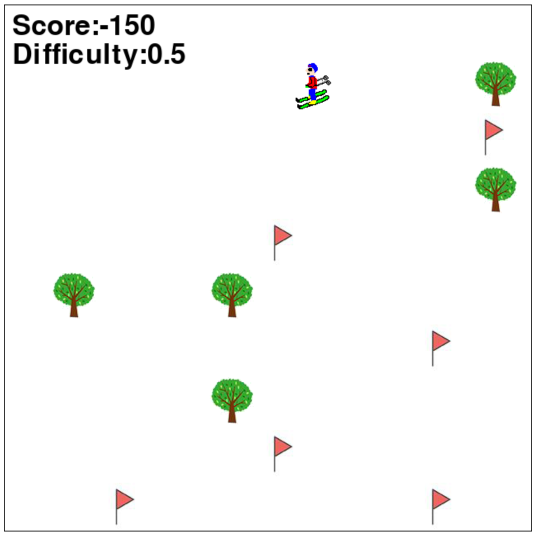

Figure 1: Game environment rendering.

(trees). The scrolling speed is set to maximum when moving in a straight line and gradually decreases when moving sideways. The game interface includes a difficulty level and a game score (proportional to the rewards provided to the agents).
Two agents, the Solver and the Generator, were involved to implement the desired behavior. The Solver controls the player's movements by predicting the player's action at every step, while the Generator generates a new map of obstacles on demand when the player needs it. Flags, on the other hand, are randomly placed in the map, and the Generator itself will generate obstacle locations accordingly, making it easier or harder for the player to reach his score maximization goal.

2.1 Representation

Both the solver and the generator are trained in an actor-critic framework, with two models each to predict a policy and a value function. While the actor and critic networks are identical except for the size of the final output layer, to handle the complexity of the task the solver and generator are designed slightly differently. For the actor model, the solver uses a simple CNN consisting of three layers to extract visual features from the input, and then a fully connected layer with a softmax final activation is in charge of predicting the probability distribution for the available actions. A preprocessing similar to (3) was used for the input, concatenating in the depth dimension 4 images converted to grayscale and downscaled, which correspond to the rendering of the scene at the previous steps. To use less correlated samples, frame skipping was used, letting the solver decide for a new action every 8 frames.
The generator was designed, as in (2), to use an auxiliary input that could parameterize the difficulty of the generated scene. A CNN extractor, as with the solver, is used to extract the visual features which are then concatenated with a second branch fed with the auxiliary input. The whole is processed with a fully connected layer that predicts values for a 10x10 grid representing map locations. In this case, the actor network input is the next proposed scene with flags already randomly placed. To handle the locations already occupied, an additional masking layer is applied before computing the softmax. The critic, on the other hand, having to judge the actor's prediction, will see the full map with flags and trees so as to correctly estimate the value of the state.

2.2 Training

The models are trained with Proximal Policy Optimization (PPO) and a self-play version in which training is done alternately between the two agents. In detail, since the generator is exposed to far fewer steps per episode than the solver (the former will collect one sample per map generation instead of the latter one per action selection), the number of episodes devoted to training the generator is double that of the solver.

PPO is a policy gradient method that belongs to the family of Actor-Critic algorithms. In practice:

1. Create a buffer of samples interacting with the environment for multiple steps: the actor network chooses the actions using learned policy and the critic network evaluates each state value.
2. Collected a buffer of trajectories the advantage function compute returns and advatages for each state in the buffer using Monte Carlo estimate or TD.
3. Compute the loss and update both networks, minimizing the cost function while ensuring a deviation from the previous policy being relatively small.

$$L^{CLIP+C+E}(\theta) = E[L^{CLIP}(\theta) - \gamma_1 L^{C}(\theta) + \gamma_2 L^{E}] \tag{1}$$

Actor-Critic methods, such as PPO, use the entropy (the measure of randomness) of the policy as part of the loss function, which inherently encourages exploration. L^E in the function above. High levels of entropy are expected at the beginning of training, since all actions may have an almost equal probability of being selected, but as the model explores actions and collects rewards, it will gradually encourage the adoption of actions that lead to higher rewards, and thus entropy will decrease as training progresses and gradually the policy begin to behave more greedily.

The key contribution of PPO is ensuring that a new policy update does not change it too much from the previous policy by using a clipped loss value. This results in less variance in training at the cost of some bias, but it ensures smoother training and also makes sure the agent does not undertake an irretrievable path of senseless actions. In the equation above is L^{CLIP} term, defined as

$$L^{CLIP}(\theta) = min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$$

where A_t is an advantage function.

The remaining element in Equation 1 is a mean square error of the value function, in charge of updating the critic network L^C, and γ_1, γ_2 are hyperparameters.

2.3 Reward

The reward shaping was thought on one hand to incentivize the player to run while avoiding obstacles without forcely make him/her collect all flags, while on the other hand the map generation should aim to make the game challenging, following the difficulty level set, but not impossible so as not to discourage the player. Each game/episode last ∼16k steps (collecting 2048 trajectories for the solver using a frame skipping value of 8) after which the currently training agent is updated. At each step:

- the solver is rewarded with +1 for each rewarding object and -2 for each obstacle encountered
- the generator is rewarded $\Sigma_{i=0}^{n}$ *r_solver*$_i$∗*diff* where $\Sigma_{i=0}^{n}$ *r_solver*$_i$ is the sum of rewards obtained by the solver while facing the generated map for n steps, and *diff* is the auxiliary input stating the difficulty value, belonging to [-1,1].
  In this way, the generator aims to make the agent perform poorer when the difficulty is high (*diff*$= -1$) and conversely when the difficulty is low (*diff*$= 1$), but since the desired behavior is to challenge the player and not strike him/her out, a large final reward is forwarded at the end of the episode: +10 if the player finishes with a positive sum of rewards or -10 otherwise.

3 Experiments

Different aspects of the pipeline were tested and evaluated in an attempt to achieve the set goals, keeping the configuration that provided the best results.
Starting from the underlying model representation and architecture, a MLP network was evaluated in place of the proposed CNN, along with a simplified state representation. This, to avoid complex spatial features, depicts the scene elements as simple squares and also provides a more machine-interpretable game interface. Anyway, since the results obtained were similar in magnitude, the simpler network (less weights) was retained.
A hyperparameter tuning step was performed as a Bayes search (using the Weights and Biases

platform) to find the best learning rate values for the actor and critic networks (0,00005 and 0.0025), as well as the batch size (64) and number of epochs to run on a single buffer of trajectories (15).
Regarding the reward shaping problem, I focused my experiments on a solver being trained on a completely randomly generated environment.

- By assigning rewards and punishments of the same magnitude, the solver assumes a much more dismissive behavior by going against obstacles in case there are rewarding objects on the same trajectory.

- To incentivize seeking and obtaining more rewards within a single game, I also tried assigning a small negative reward in the case of consecutive steps without obtaining rewards, but this did not particularly affect the agent's final training result.

- Finally, assigning large final rewards or punishments to the solver, as in the case of the generator, resulted in very large imbalances in losses and overall worse performance, so I did not adopt it.

The final implementation leaves more freedom for the two agents to exploit useful strategies to achieve their respective goals, without forcing risky choices on the solver and adapting more flexibly to the rules of the game.

4 Results

To demonstrate the usefulness of the framework, I evaluated two solvers by comparing their results: one, called baseline solver, was trained for 150 episodes in an environment generating scenes randomly, while the other, the expert solver, was trained in pairs with the generator as explained in 2.2 using the same parameters and an equivalent number of episodes. Both solvers were tested in both environments, to consider both the results obtained in a "known" situation, which is the environment creating maps with the same probability distribution used during training, and the other considered as a validation test.
From Fig.2 we see the average reward per step obtained by the two solvers in 30 episodes. The maps were generated by the Generator set at different levels of difficulty depending on the test. The average reward per step is obtained as the total reward of the solver in the episode divided by the number of steps. It can be seen that the expert solver has an advantage in this environment: having already been exposed to the generator it is able to perform equally well at all levels of difficulty. In contrast, weakness and poor performance at a higher difficulty level can be seen in the performance of the baseline solver, which on average gets progressively worse by increasing the difficulty level.
I hypothesize that if the expert solver really learned to be more proficient, this would also be reflected in the environment with completely randomly generated maps. Indeed, Table 1 shows that, by reporting the game score obtained by the player on average over 30 episodes. The expert solver performed excellently by getting more than 300 points over the baseline. In addition, from the average number of objects hit by the player in a single episode (trees and flags), a greater ability of the player to choose objects is observed, considering that a "take all" approach could still have yielded good results being the number of flags generated greater than the number of trees.
Performance can also be judged qualitatively by observing the two solvers play. The basic solver often chooses wrong trajectories, which in order to collect a few points lead him to loose more than he gains; both have learned that by playing on the side of the map they can easily avoid obstacles in their paths by simply dodging them and returning to position, but the basic solver by moving too late or abandoning the spot is unable to make the most of the strategy. The expert solver, on the other hand, is very good at selecting actions to dodge obstacles and taking combinations to reach objects far away in time but visible in the map; in more difficult situations he moves the player to safer positions on the sides of the map so as not to risk losing excessive points.
The generator demonstrates skill in creating challenging maps, as seen in the baseline evaluation, making predictions in accordance with the difficulty setting. It often places trees in front of or behind flags making it difficult to reach them without also hitting the obstacle. In addition, since the sampling of obstacle locations is done through the generator policy, predicting higher probability values for a single location allows to decrease the number of obstacles.

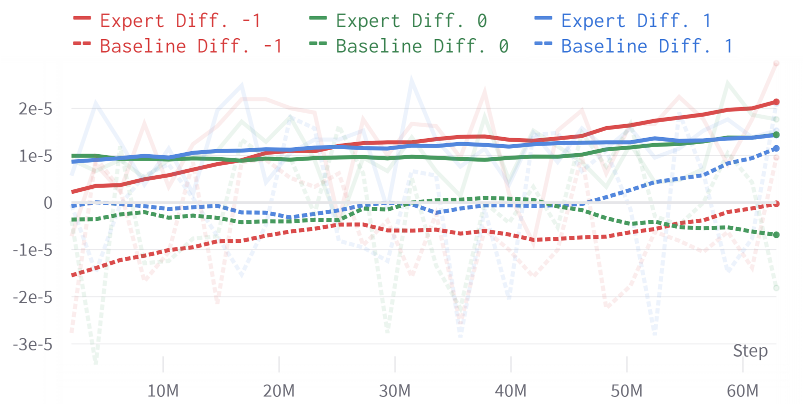

Figure 2: Average reward per step obtained by expert solver (agent trained with a generator) and baseline solver (trained with randomly generated maps) acting for 30 episodes in an environments governed by a generator set at different levels of difficulty. The results collected are smoothed with a running average to highlight the average range of results collected.

Table 1: Games score and number objects hit averaged over 30 episode both for expert solver and baseline solver. Results collected on interactions with both types of environments, one governed by a generator agent and one randomly generating maps. The game score obtained by the player is proportional to the rewards obtained by the solver but scaled up for sake of game experience.

	Difficulty	Game score		Total objects hit	
		Baseline	Expert	Baseline	Expert
Random map		697.5	1032.5	2792	2693
	1 (easy)	-37.5	642.5	2718	2293
Generator agent	0 (medium)	-87.5	528.3	2662	2196
	-1 (hard)	-290.8	554.2	2862	2294

5 Conlusion

The project shows how an agent trained on well-designed situations to challenge him is able to outperform a similar agent exposed only to randomly generated maps. The results were confirmed by both qualitative and quantitative analysis, suggesting that this methodology aids learning by identifying more complex trajectories and more carefully choosing actions and spots where to move. A generator agent trained in pair with a solver was designed to generate creative maps in line with a parameter of difficulty set. It should also be noted that the choice to train such an agent with an improving solver was crucial to the end result, as otherwise the generator would have degenerated to almost deterministic choices, focusing on the weaknesses and blind spots of the current solver.
Although the framework is quite general and adaptable, it has only been tested on the described game environment, so further extension could include training in other worlds with similar rules to assess the generalization ability of the agents.

References

[1] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. PCGRL: procedural content generation via reinforcement learning. In Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'20). AAAI Press, Article 14, 95–101.

[2] Linus Gisslén, Andy Eakins, Camilo Gordillo, Joakim Bergdahl, and Konrad Tollmar. 2021. Adversarial Reinforcement Learning for Procedural Content Generation. In 2021 IEEE Conference on Games (CoG). IEEE Press, 1–8. https://doi.org/10.1109/CoG52621.2021.9619053

[3] Mnih, Volodymyr Kavukcuoglu, Koray Silver, David Graves, Alex Antonoglou, Ioannis Wierstra, Daan Riedmiller, Martin. (2013). Playing Atari with Deep Reinforcement Learning.

[4] Tests and hyperparameter search available here: https://wandb.ai/johnminelli/EnvKnob