Snake Game 2D – Reinforcement Learning

Faculty of Computer Science and Engineering
Agent-Based Systems – 2022/2023
Ace Gjorgjievski – 201183

Abstract – As we can all see that the technology is spreading massively, video games are beside them, but would you consider watching a machine playing a video game? This research is about an "old school" snake game 2d coded in python with pygame library and all of that at the end the game plays itself to eat the food and get longer – The Reinforcement Learning process. The graphical representation is made with pygame, integrated with gym-compatible environment, and trained with stable-baselines3.

Keywords: *video game, machine, snake game 2d, python, reinforcement learning*

I. Introduction

Reinforcement learning is a massive area in artificial intelligence which gains a huge momentum. The agents are exploring and learning about the environment and gaining positive or negative rewards according to their taken actions. The core application of the reinforcement learning is sort of a compelling adventure, intelligent behaviors in the digital environments. Upon this, we dive into the 2d games, particularly the "old school" snake game 2d.

The Snake Game is sub-genre of action video games where the player maneuvers the end of a growing line, often themed as a snake. The player must keep the snake from colliding with both other obstacles and itself, which gets harder as the snake lengthens. It originated in the 1976 two-player arcade video game Blockade from Gremlin Industries where the goal is to survive longer than the other player. [1]

Now, in the 21st century, where the video games are literally everywhere, the snake game will play itself. How? By exploring the environment, hitting the walls, hitting itself, or better said through own wins and failures to achieve better results in the end.

The game is created with python in pygame library, integrated with the gym library to present the abstracting of the snake game and trained with stable-baseline3 which is a set of reliable implementations of reinforcement learning algorithms in PyTorch, built on top of OpenAI Gym.[2] Furthermore, we will discuss the detailed implementation of the game.

II. Snake Game Implementation

First, the game was coded in python, with the pygame library, with the standard snake game rules: grid size (width x height), snake list, food and other parameters.

Next, the game inherits from the gym library in order to implement the necessary functions provided from the gym library for implementing reinforcement learning:[3]

- **step(action)**: updates the environment with actions returning the next agent observation;
- **reset()**: resets the environment to an initial state;
- **render(render_mode)**: renders the environment in order to visualize what the agent sees, typically the render_mode is set to 'human'
- **close()**: closes the environment

When it comes to different algorithms in order to train the agent model, one of the most prominent is Q-Learning. It is the fundamental approach to find optimal decision-making strategies. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the current value and the new information.[4]

$$Q^{new}(S_t, A_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \Big(\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}} \Big)$$
$$\text{new value (temporal difference target)}$$

Algorithm 1: Q-Learning

The parameters are the following:

- $\mathbf{R_{t+1}}$ – reward received when moving from the state S_t to the state S_{t+1}
- $\boldsymbol{\alpha}$ – learning rate $0 < \alpha \leq 1$
- $\mathbf{(1 - \alpha) * Q (S_t, A_t)}$ – the current value (weighted by one minus the learning rate)
- $\boldsymbol{\gamma}$ – discount factor
- $\mathbf{S_{t+1}}$ – next state after taking action a in state s
- **a** – the action that maximizes the Q-value in the next state S_{t+1}

There are several rewards and penalties in the snake game implemented for a better navigation of the agent (snake) in the environment, like:

- **Death**: if the snake is dead, the penalty is -100.
- **Eating an apple**: if the snake has eaten an apple, the reward is 10.
- **Distance**: if the snake is closer to the apple, a reward of 1 is given and if the distance increases, a penalty of -1 is given.
- **Wasting time**: punishment is calculated based on the number of timesteps passed since the last apple has been eaten.

The process of training is implemented by stable baseline3, using the PPO (Proximal policy optimization) algorithm which combines ideas from A2C (Advantage Actor-Critic) and TRPO (Trust Region Policy Optimization). The main idea is that after an update, the new policy should not be too far from the old policy.[5]

In this training we will evaluate and see the PPO and A2C algorithms separately, compare their final training values from the agent model and see who is better. In this training I used several intervals to visualize the process of learning. The intervals were: 50000, 100000, 250000, 500000, 1000000 steps.

rollout/	
ep_len_mean	28.1
ep_rew_mean	-101
time/	
fps	19
iterations	200000
time_elapsed	51072
total_timesteps	1000000

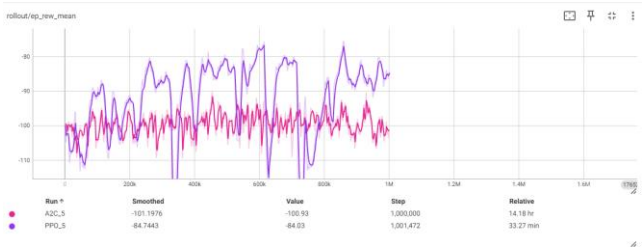
train/	
entropy_loss	-0.465
explained_variance	0.442
learning_rate	0.0007
n_updates	379999
policy_loss	1.36
value_loss	38.6

Figure 1: Key metrics from training snake game RL model

In the following *Figure 1* we can notice several metrics when training the snake game with reinforcement learning using the Proximal Policy Optimization (PPO) or Advantage Actor-Critic (A2C) algorithm. The **rollout statistics** give an overview of the agent's behavior performance during the early stages of training, providing valuable information about how well the agent is adapting to the environment and what kind of reward it is obtaining on average per episode, like episode length mean and episode reward mean. **Time statistics** typically provide information about the time-related aspects of the training process, like frames pre seconds, iterations, time elapsed etc. The metrics of the training process can be seen in the **training statistics** which focus on the learning algorithm's performance and behavior during training. Some of the metrics are entropy loss, learning rate, policy loss etc.

III. Visual Metrics on TensorBoard

When the model has been trained, we can visualize the key metrics on tensorBoard. On the following table we can see the metric of the episode reward mean after 1 million steps:



We can see that here PPO wins ahead of A2C, which PPO is a combination of A2C and TRPO. A2C uses workers (parallel agents) to gather experience from the environment. On the other hand, the TRPO, in some way, has a controlled way of updating the policy, preventing large deviations to maintain stability during the learning process.

IV. Conclusion

In summary, this was a research project for my Agent-Based System course where I had to make a game that learns to play by itself (learning the machine to play the game). The process was a little bit exhausting and fun because, not only in this course, but in every and each one you learn, you learn just a tiny amount of it, or better said, just enough to dive into the realm of the real knowledge that sustains there. It was fun at the same time when you learn new things in the sphere of artificial intelligence where can see how the machines learn and see the environment, like I noticed one comment on the social networks that says: "First we enjoy playing video games, then we enjoy watching other people playing video games, next we enjoy watching machines playing video games".

V. Acknowledgements

The source code of the snake game 2D – RL can be found here.

VI. References

[1] https://en.wikipedia.org/wiki/Snake_(video_game_genre)
[2] https://stable-baselines3.readthedocs.io/en/master/
[3] https://gymnasium.farama.org/api/env/
[4] https://en.wikipedia.org/wiki/Q-learning
[5] https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html