SNAKE AI

Stanisław Giziński
Uniwersytet Warszawski
sg385513@students.mimuw.edu.pl

Piotr Książek
Uniwersytet Warszawski
pk371141@students.mimuw.edu.pl

Maciej Twardowski
Uniwersytet Warszawski
mt386458@students.mimuw.edu.pl

ABSTRACT

We present an implementation and propose a solution to a clone of a classic Snake game as a Gym environment. We briefly describe the process of adding a custom RL environment to OpenAI Gym toolkit. We build our Agent on the classical Deep Q Learning algorithm and describe the outcomes of various experiments we took to improve its performance in the Snake environment.

***K*eywords** Snake · OpenAI Gym · Reinforcement Learning · Deep Q Learning

1 Introduction

The idea of the project comes from OpenAI's Request for Research 2.0. Given that none of the Authors had any experience with Reinforcement Learning, we decided on limiting the problem to the task of implementing and solving a classical, singleplayer Snake environment.

2 Implementing Snake

We wanted to write our own Snake clone to have full control over the way the game was implemented. It has proven useful in the future, see Section 3.4.1 and Section 4.2.

2.1 Creating the OpenAI Gym Environment

The process of adding an environment was described by Ashish Poddar in *Making a custom environment in gym*[1], and we won't focus on that part.

3 Solving the Environment

3.1 Preparations and preconditions

We decided on choosing an 8x8 map. It's arguably harder to play Snake on a smaller board. At the same time, however, the Agent is more likely to randomly collect a fruit, which limits the sparse reward problem and makes the exploration process a lot easier. At each step, the agent would receive an observation as a 2D array with integers representing walls, Snake's body, Snake's head, and the fruit. Additionally, we would pass them 2 previous observations, to make it possible to determine Snake's current direction. The agent would then select one of 4 possible actions - move up, left, down, right. Initially, the Snake was allowed to turn back in place and kill itself in the result (as it would collide with itself). Due to the requirements of high computing power the training process was limited to 100k-200k episodes.

3.2 Deep Q-Learning

Deep Q-Learning has been described in Mnih's paper [2]. The core idea is to combine a classical approach called Q-learning with a Neural Network. Q-learning aims at discovering a function $Q : S \times A \rightarrow \mathbb{R}$ that returns a future discounted reward for each $(state, action)$ pair. The Agent will then select the action that maximizes the reward. We use a Neural Network to approximate Q.
Q has several properties that we can use.
One is given by the bellman equation:

$$Q(s, a) = r + \gamma max_{a'} Q(s', a') \tag{1}$$

And the classic update formula at step t:

$$Q_{t+1}(s_t, a_t) = (1 - \alpha)Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q_t(s_{t+1}, a)) \tag{2}$$

The way Q values are updated based on the rewards observed by the agent is controlled by two parameters: α and γ.

- α influences the size of changes in the estimated Q values. It determines how much we allow a single observation to affect our idea of the Q value for that state-action pair. It's often referred to as Q-learning rate. Note that this is **not** the learning rate used in the model's optimizer. The model has its own learning rate, independent of α, that controls the size of changes to the weights of the model during training.
- γ is the discount factor, determines how important future rewards are

Algorithm 1 Replay

{$Q' : S \rightarrow A \times \mathbb{R}$ is our Q function approximation, returns pairs $(action, value)$ for all possible states}
input batch := $\emptyset$
target batch := $\emptyset$
for state, action, reward, next state, done in memory sample **do**
    target := reward + (1-done) $\cdot\gamma\cdot$ $max(Q'$(next state))
    current Q'(state) := Q'(state)
    target Q'(state) := Q'(state)
    target Q'(state)[action] $\leftarrow$ (1-α)$\cdot$ Q'(state)[action] + $\alpha\cdot$target
    input batch $\leftarrow$ input batch $\cup$ {current Q'(state)}
    target batch $\leftarrow$ target batch $\cup$ {target Q'(state)}
end for
fit Q' using (input batch, target batch)

3.3 Exploration

After we initialize our model, we start with a pretty much random Agent. Over time, however, the Agent will learn and follow some policy. We cannot ensure that this isn't a local minimum. To help the Agent get a better overview of the situation, we use the exploration rate ϵ. It is set so that $0 \leq \epsilon \leq 1$ and $\epsilon \longrightarrow 0$ over time. In each step, the Agent will pick a random action with probability ϵ or act accordingly to its policy with probability $1 - \epsilon$.

We implemented a Replay Buffer to store the experiences as suggested by DeepMind [4]. The Agent will collect observations and store them in the Replay Buffer. Observations are then randomly sampled and used to train the Agent.

3.4 Challenges

3.4.1 Self-termination

Initially, in the scheme with 4 possible actions, the snake was allowed to kill itself at any time by pressing the button opposite to its current direction. Even though the Neural Network approximated the Q value of that action correctly, the Agent would still often terminate by picking that action randomly because of the exploration rate. We fixed it by simply blocking that movement in the game's implementation. After this change, we were able to produce our first working Agent, referred to as Base in Table 1.

3.4.2 Exploding Q-values

We noticed that the Q values would sometimes increase out of control. We've diagnosed the problem as a **feedback loop**. In Algorithm 1 we use the output of the model to calculate the target value. This could cause trouble if for some reason the Agent started wrongly judging some moves as extremely valuable. To combat this problem we set a small α (Q-learning rate) ≤ 0.1. We also use a separate Target Network, as suggested by DeepMind [2, 3]. It has proven to improve learning stability at the cost of slightly longer training time. We've tried two variants of this method since there are two ways of updating the Target Network's weights. The first method simply copies the weights of the Online Network after each n steps. A bit more interesting solution, proposed by Lillicrap et al. in *Continuous control with deep reinforcement learning* [3], called 'soft update', sets the weights of the Target Network to a weighted average of its and the Online Network's weights after each training step. We did not notice a significant difference between those two methods in our case, they have, however, produced our second best agent. See Target Agent in Table 1.

3.4.3 Loops

The Agent would sometimes start moving in a loop. The Q value of each $(state, action)$ pair is deterministic. If the Snake, starting at a certain state, picked an action that resulted in it returning to the same state without changing the environment (consuming the fruit), it would pick the same action and perform the same move sequence over and over again. During training, the Agent would sometimes make a random move, which allowed them to break the loop, but it wasn't the case once the training's been finished. We try to solve this problem by increasing exploratory rate and introducing penalty in reward to each move, see Section 4.2.1. Note that the Target Network approach (3.4.2) also shows a significant improvement.

4 Experiments

4.1 Modified Q updates

We decided to set a constant, negative Q value t for each $(state, action)$ pair resulting in termination. In Algorithm 1, if $state$ is terminal, we set $target$ to a fixed value t. This implementation was suggested in Mnih's paper [2] and showed some improvement over our Base Agent.
See ModQ Agent in Table 1.

4.2 Limiting action space

There are at least two ways of implementing controls in Snake. One that comes naturally is to give the player 4 buttons (up, left, down, right) which determine which way they want their Snake to go in the next move, and if they don't press any button, the Snake will continue moving in the same direction. Note that this means two arrows always do nothing - pressing the arrow corresponding to Snake's current direction won't change anything, but neither will choosing the opposite arrow - the snake cannot turn back in place. A less lossy solution is giving the player (or the agent) only three options in each move - turn left, keep the current direction, turn right. It may seem like a minor difference, but the results show that the Agent trained on a smaller action space achieved higher results much faster. A possible explanation is that in this control scheme the probability distribution of action selection in the exploration phase is uniform. In the previous diagram, the movement that did not change direction was twice as likely.
See 3Actions Agent in Table 1.

4.2.1 Move penalty

With this agent we tried another method of avoiding the looping problem. We set a small penalty for each move. The results were similar to those of the Target Agent, a smaller looping rate at the expense of slower training.
See 3ActionsMP in Table 1.

5 Revisit

We decided to come back to this project and rewrite it using Pytorch since it would allow us for more flexibility with experiments than Keras. We wanted to try some policy related methods. Since we already had a working Q-learning algorithm, we decided to implement the Actor-Critic algorithm.
The agent achieved decent results quite quickly (fewer than 20000 episodes) with an average length just below 14 and no loops. However further training became problematic, the progress was very slow, and after some time the snake would occasionally loop.

Table 1: Agent comparison

Agent version	Average length	Loops	Number of episodes
Base	8.07	19%	100k
ModQ	10.63	8%	100k
3ActionsMP	12	5%	125k
Target	12.44	5%	190k
3Actions	15	10%	125k
ActorCritic	17	<1%	100k

We've increased the number of layers and their sizes and also introduced penalties for loops. In each episode we would store the hashes of states that have already been observed, and if a state had reoccurred, the agent would be terminated. It makes sense, because the longer the snake, the less likely it is for a random agent to loop. The probability of looping while acting randomly decreases exponentially in relation to length. With $p \sim 0.015$ of terminating a base len. 3 snake, it doesn't hurt the exploration process, while it provides valuable feedback for looping agents.

6 Summary

We implemented our own Snake environment and used it to learn and demonstrate some core concepts of Reinforcement Learning. We achieved a significant improvement over our first Agent by facing and solving many problems typical for this field.

References

[1] Ashish Poddar. Making a custom environment in gym `https://medium.com/@apoddar573/making-your-own-custom-environment-in-gym-c3b65ff8cdaa`

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning *arXiv:1312.5602*, 2013.

[3] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, Daan Wierstra. Continuous control with deep reinforcement learning *arXiv:1509.02971*, 2015.

[4] Max Jaderberg, Wojciech M. Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C. Rabinowitz, Ari S. Morcos, Avraham Ruderman, Nicolas Sonnerat, Tim Green, Louise Deason, Joel Z. Leibo, David Silver, Demis Hassabis, Koray Kavukcuoglu, Thore Graepel. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning *arXiv:1807.01281*