REWARD SHAPING ON ATARI-SNAKE WITH DISTRIBUTED A2C

Pavan Bhargav, Sai Ganesh, Lakshmi Narayana & Avinash B.
1600500{76,78,80,82}
Department of Computer Science
Indian Institute of Technology, Bombay

ABSTRACT

In this paper, we explore various methods to maximize the total score attained by snake[1] in the classic Atari game. We implemented the distributed A2C algorithm with different rewarding techniques, and trained it using two different model architectures namely convolutional and feedforward. We tested each of these on four different reward techniques - basic food reward, negative reward on death, path optimization reward and timeout mechanism.

1 INTRODUCTION

Reinforcement learning has been applied to play simple games decades ago and extensive research has been done to improve the learning algorithms from basic MDP methods to more complex and state-of-art techniques like A2C, A3C, D-DQN, etc. What we wish to explore in this paper is rather the less researched area of effect of reward shaping on the performance of learning agents. With this idea in mind, we decided to consider the base-line as an implementation of a classic Atari game with scope of improvement using reward shaping. The game contains a 11x11 grid with snake of initial size 3 with food appearing at random positions in the grid.

Snake can choose any of four directions at each step to eventually get the food after which it's size increases by one while making sure that it doesn't collide into itself or one of the edges. In the following section, we explain the algorithm for training the agent. You can skip the section if you are already familiar with distributed A2C algorithm.

2 ALGORITHM

In this paper we adopted to Advantage Actor-Critic (A2C) model for agent learning as it is currently one of the most powerful algorithms in deep reinforcement learning. As our main focus is on reward shaping we didn't focus much on implementing the algorithm from scratch and referred an implementation[1] which is our baseline where A2C is implemented in a distributed fashion using pytorch.

2.1 A2C

The idea behind the actor-critic algorithm is that we train an actor network to map from observed states to a probability distribution over actions i.e. the policy. The aim of this network is to learn the best actions to take in a particular state. At the same time we also train a critic network that maps from states to their expected value i.e. the sum of future rewards expected after being in this state. The aim of this network is to learn how desirable it is to be in a particular state.

- The Critic measures how good the action taken is (value-based) V(s)
- The Actor outputs a set of action probabilities the agent can take (policy-based) Q(s,a)

[1]Initially we submitted a project proposal of implementing a3c on pinball game but as it lacked novelty, we chose to implement atari snake and improve it with our own reward system

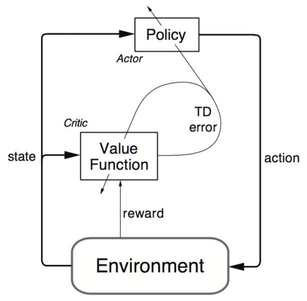


Figure 1: figure from Sutton and Barto

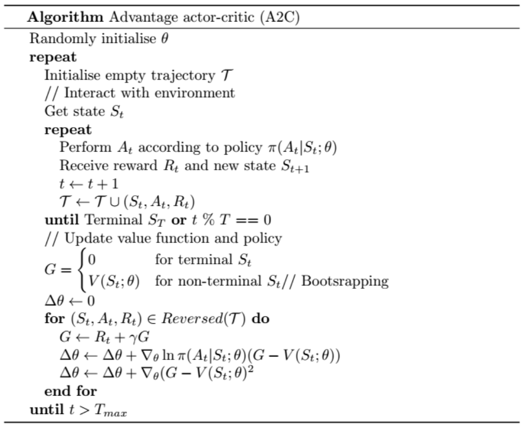
Basically the actor takes as input the state and outputs the best action. It essentially controls how the agent behaves by learning the optimal policy (policy-based). The critic, on the other hand, evaluates the action by computing the value function (value based). Those two models participate in a game where they both get better in their own role as the time passes. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately.

The Advantage is how the Critic tells the Actor that it's predicted Q-values from the ANN are good or bad. It calculates the policy loss. This is calculated through the Advantage equation.

$$A(s,a) = \underbrace{Q(s,a)}_{\text{q value for action a in state s}} - \underbrace{V(s)}_{\text{average value of that state}}$$

Figure 2: Advantage

2.2 Overall algorithm



```
Algorithm Advantage actor-critic (A2C)
Randomly initialise $\theta$
repeat
  Initialise empty trajectory $\mathcal{T}$
  // Interact with environment
  Get state $S_t$
  repeat
    Perform $A_t$ according to policy $\pi(A_t|S_t;\theta)$
    Receive reward $R_t$ and new state $S_{t+1}$
    $t \leftarrow t+1$
    $\mathcal{T} \leftarrow \mathcal{T} \cup (S_t, A_t, R_t)$
  until Terminal $S_T$ or $t \% T == 0$
  // Update value function and policy
  $G = \begin{cases} 0 & \text{for terminal } S_t \\ V(S_t;\theta) & \text{for non-terminal } S_t \text{// Bootsrapping} \end{cases}$
  $\Delta\theta \leftarrow 0$
  for $(S_t, A_t, R_t) \in Reversed(\mathcal{T})$ do
    $G \leftarrow R_t + \gamma G$
    $\Delta\theta \leftarrow \Delta\theta + \nabla_\theta \ln \pi(A_t|S_t;\theta)(G - V(S_t;\theta))$
    $\Delta\theta \leftarrow \Delta\theta + \nabla_\theta(G - V(S_t;\theta)^2$
  end for
until $t > T_{max}$
```

Figure 3: Algorithm

Here we have sketched out the overall algorithm. This is the synchronous version of the famous asynchronous advantage actor-critic algorithm (A3C). The difference is that in A3C the parameter updates are calculated in many worker threads separately and used to update a master network which the other threads periodically synchronize with. In batched A2C the experience from all workers is combined periodically to update the master network hence we call it a distributed learning.

The reason that A3C is asynchronous is so that differences in environment speeds in different threads don't slow each other down. However in our snake environment the environment speeds are always exactly equal by design so a synchronous version of the algorithm makes more sense.

3 Reward-Shaping

Reward mechanisms have a key role in improving the learning ability of an agent and thereby maximising score and/or survival time in the environment. Many factors of the environment have a part in deciding the maximum achievable score of agent and therefore each of them is a potential feature that can be used in reward mechanisms. In the current scenario of Atari Snake, these features can be

- Avoiding collision with itself (self-collision)
- Avoiding collision with edges(edge-collision)
- Minimising distance from head to food
- Reaching food as fast as possible

We implemented each of these reward mechanisms as following:

3.1 Baseline

The base-line implementation is taken from a Medium article[2]. The agent is rewarded +1 on capturing food and no positive/negative rewards for any other interaction with the environment.

3.2 Punishment on death

As we can see from the previous mechanism, it doesn't consider the actions of self-collision and edge-collision as a negative impact on agent's learning. By giving the negative reward in case of these types of collisions, we can guide the agent to avoid them. Reward mechanism is +200 in case of capturing food, -100 in case of death, -1 for each step taken. The negative reward which is given for every step helps in reaching food faster than previous version without taking any unnecessary steps.

3.3 Path optimization

In this technique,in addition to previous reward types, the relative position of food is calculated with respect to head. Every action that moves the head away from food is penalized with 1/L, where L is the current snake size. As the length of the snake increases, it gets tougher to reach food just by moving towards it as the snake body can be an obstruction. Thus the negative reward is minimized with the increase in snake size allowing it to consider longer paths thereby avoiding self-collisions. Compared to previous version, this one is stricter way of forcing the head to reach the food.

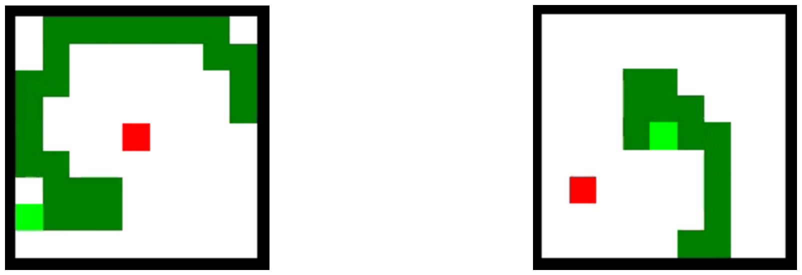
3.4 Timeout Strategy

In this method, if snake doesn't obtain food in "p" steps (where p = 0.7*L+10, L is length of snake at the instant[3]) after consuming a food recently, it will be penalized with 1/L for every further step until it reaches the food again. Here "p" term is proportional to L because the snake should be given more time as it's length increases and it gets tougher to reach the food. This mechanism forces the agent to obtain food as early as possible compared to previous version.

4 EXPERIMENT SETUP

We used a vectorised and massively scalable implementation of a snake-like game as a reinforcement learning environment. Comparisons are made using the model architecture - Convolutional agent with 11x11 grid (9x9 moving area if padding is omitted). Each convolutional model run for approximately 5hrs to complete 250e6 steps on Intel i7 12 cores 8GB CPU and 1050Ti 4GB GPU.

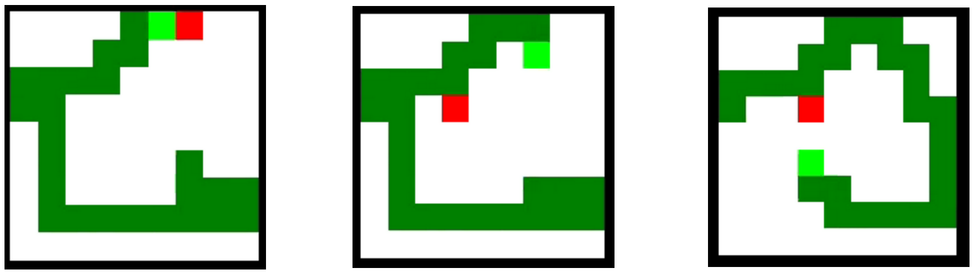
5 RESULTS



(a) edge-collision (b) self-collision

Figure 4: Collisions after training with base-line model

In the baseline model the low scores are due to early edge and self collisions which are occurring because of not penalizing them.

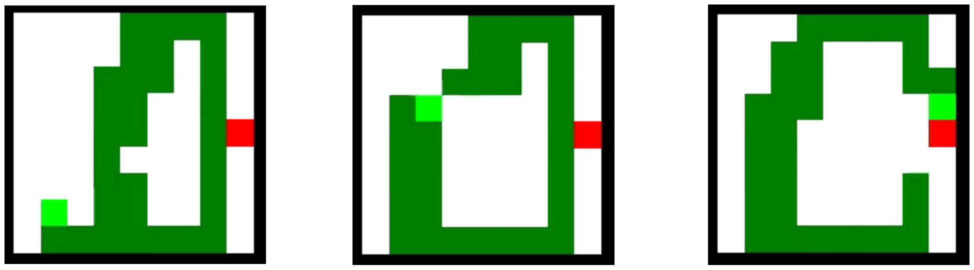


(a) just before new food appears (b) new food appears (c) takes a long path to eat the food

Figure 5: Non optimized path of snake with 2nd reward technique

Using 2nd reward metric, it avoids self and edge collisions but we can see that after eating food in 1st image it goes on a long path from 2nd to 3rd to get the next food.

To avoid long paths we imposed a 1/L punishment for an L length snake on moving away from food in the 3rd reward system.



(a) almost impossible to get food (b) starts thickening itself (c) gets the food

Figure 6: Snake trying to avoid self-collision when it's size is large

An interesting observation here was in 1st image it was almost impossible for the snake to get the food without self collision, but it learned to decrease the effective length by piling up into 2 width thickness and was able to grab the food by moving from 2nd image case to 3rd.

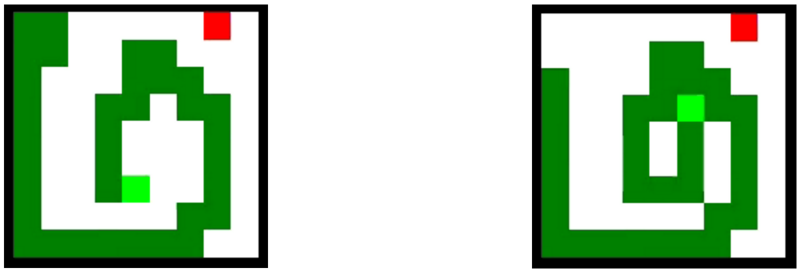

(a) wrong action (b) eventual death

Figure 7: Flaw in agent behaviour

In the 4th observation the implementation of timeout mechanism makes the movement away from the food very difficult, even for it to survive. We have observed that the self collision rate has increased than the previous mechanisms.

6 Observations

The total time taken to complete 250e6 steps in case of 1st and 2nd versions is 4hrs50min and for 3rd and 4th cases, it took almost 5hrs5min. We can conclude that the total time taken is almost the same for all versions.

We first trained using 9x9 grid and the improvements are not as expected even with promising reward shapings. We came to a conclusion that it's because of the small area for the snake to improve itself and thus we shifted to 11x11 grid.

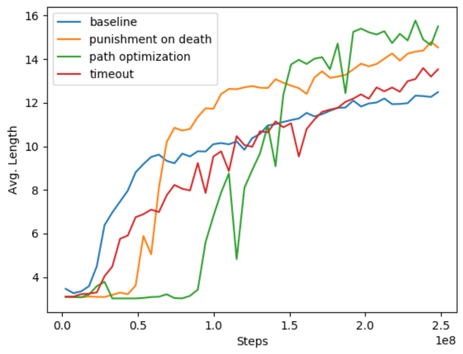


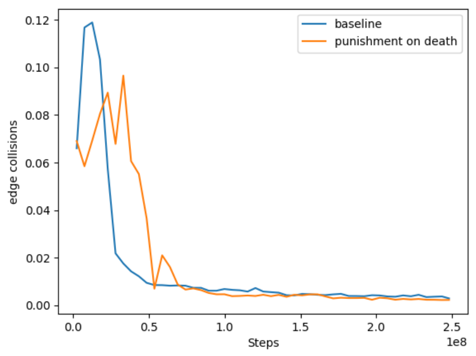
Figure 8: Plot of average length vs steps for all reward mechanisms

Average snake size is highest for "path optimized" reward shaping followed by "punishment on death" and "timeout". It was expected to be maximum in the case of timeout but it focuses too much on food gathering and succumbs to self-collision as shown in the figure 8.

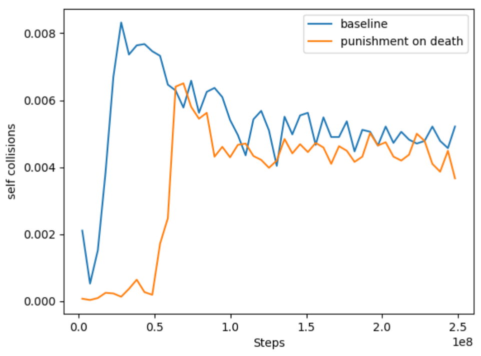
The better performance of punishment on death case over base-line can be attributed to the decrease in self-collision and edge-collisions that can be inferred from figure 9a and figure 9b.

From figure 10, we can observe that there is a slow increase in self-collision. This can be explained due to increase in average length of snake which increases the probability of hitting itself.

The peak of edge-collisions in the same figure can be reasoned with the fact that initially the agent spends more time exploring and starts performing better after sometime.



(a) edge-collisions vs steps



(b) self-collisions vs steps

Figure 9: Comparison of collisions for base-line model

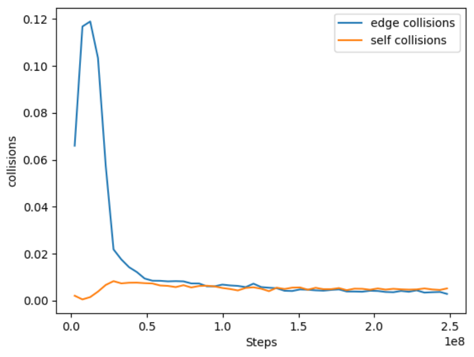


Figure 10: edge and self collisions for base-line model

Though the average length during training is coming out to be around 15-16, while rendering the average lengths went significantly higher than training and the max length obtained was 37.

Comparison of average values	
Reward Model	Average SIze
Baseline	12-13
Punishment on death	14-15
Path optimization	15-16
Timeout strategy	13.5-14

Feedforward learning has a better initial pickup compared to convolutional agent but the final convergence score is more for the latter.

7 Conclusions and Future work

As we have seen, reward shaping has a major role in the score attained by the agent in any game and

The reward-shaping can further be improved if the agent is trained in such a way that at each step it obtains a little negative reward if it moves in the direction along which there's a chance of self-collision.

We believe that with further increase in the grid size (which we couldn't do because of resource constraints) the improvement due to reward shaping increases even more.

References

[1] Baseline implementation,
https://github.com/oscarknagg/wurm/tree/medium-article-1

[2] Medium post by Oscarknagg

[3] Timeout mechanism,
http://www.ntulily.org/wp-content/uploads/conference/Autonomous_Agents_in_Snake_Game_via_Deep_Reinforcement_Learning_accepted.pdf