

Using Reinforcement Learning to Play Hungry Geese

Wang Chao¹ He Zhenfeng¹ Tian Xueyan¹

¹National University of Singapore

Background Review and Problem Description

Hungry Geese is a Kaggle competition which requires 4 players to participate. It generally adapts the rules of famous video game Snake, where each player acts as a snake which contain a head, a body and a tail.



Initially, a snake only has a head(which can be also regarded as a tail) without a body, and it will be randomly initialized on a position of 7*11 board. The snake needs to keep moving to reach a position of food thus its body will grow up. Once a food is obtained by a snake, another food will appear in the random position of the board. However, if the snake's head contact any part of a snake(including itself), it will be eliminated immediately. After running for 200 rounds, those surviving snakes will be rewarded based on its length. For those snakes being eliminated, they will be penalized based on the chronological order (those snakes who are eliminated earlier which be penalized heavier).

Simple Heuristics to Reinforcement Learning

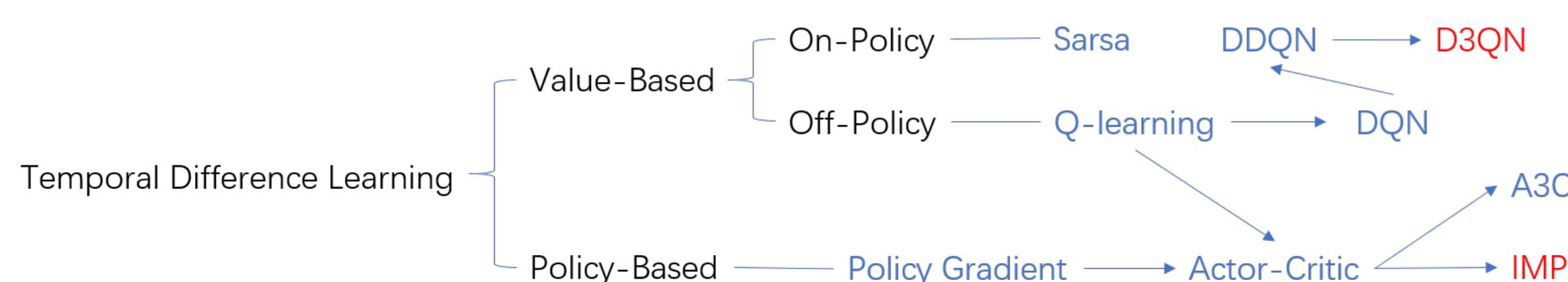
Traditionally, we could design the geese to follow specific heuristics, for instance, a greedy geese will search for food that has the minimum distance to its head and move toward it. Below we present a few heuristics that perform well on the hungry geese task.

- Risk averse greedy: Mark all cells that can be dangerous on the next step as obstacles; Find the shortest route to the nearest food item
- Flood-fill-based BoilerGoose: Calculate what is reachable for all next possible steps using flood fill algorithm; Select best action from flood-filled results; Maintain a certain length

Reinforcement learning is a technique for learning the best action in any state by trial and error, and it may be useful in solving our problem. One of the most important aspects of conducting and implementing efficient reinforcement learning algorithms to play the game is to encode the task. Each state encodes the information for (board, action, status, length), while each environment encodes board (more details for board encoding are discussed in experiment section). One important point to note is that we also encode a virtual body that is in the reverse position of the last step, such that the agent is not going to hit itself.

Reinforcement Learning Method Overview

The image below shows an overview of some major Reinforcement Learning Algorithms under subsection of temporal difference learning. In this project, we spend most of our time on the D3QN algorithm with its several variants. We also implement the IMPALA algorithm to see its potential.



D3QN Algorithm

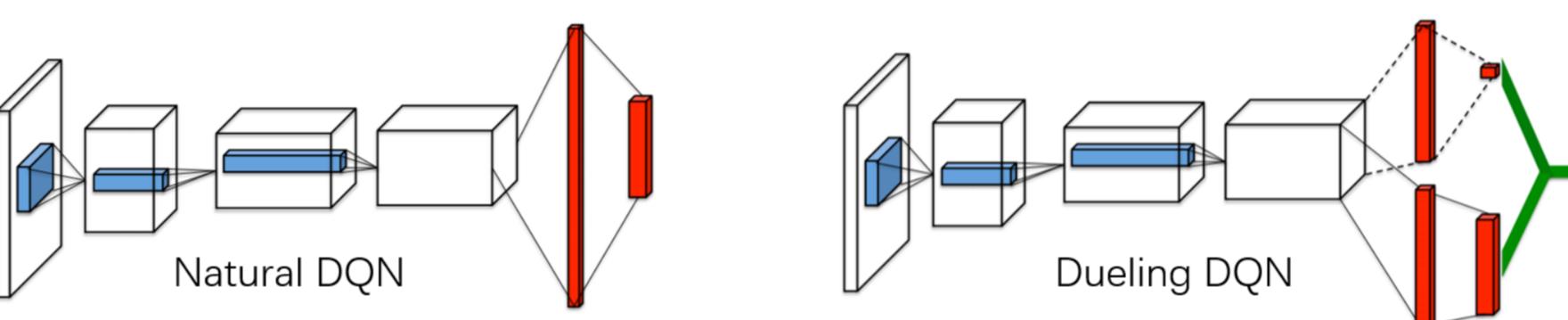
This algorithm is a combination of three different variants of DQN: 1) Double Deep Q Networks, 2) Prioritized Experience Replay DQN, and 3) Dueling Deep Q Networks.

Advantage of these variant:

Double Deep Q Networks: In natural DQN, the target Q value calculated is based on the action selected by the target Q network, which leads to the over estimation of the current state. To solve the problem, DDQN calculates the target Q value based on the current Q network. This kind of prevents over estimation by decoupling the action selection and action evaluation.

Prioritized Experience Replay DQN: In natural DQN, we have an uniform experience replay buffer to store the past game. However, different experiences may have different importance scores. A experience with a higher TD-error means the model is worse in such a state and can learn more. So, this method will assign each experience a weight by its TD-error. And, during training, samples in the buffer will be drawn for retraining considering the probability calculated using the weight.

Dueling Deep Q Networks: This method improves naive DQN by optimizing its model structure.



Natural DQN only has one Q network, $Q(s)$. In Dueling DQN, it uses the sum of V network and A network $V(s) + A(s)$ to calculate Q value. V network stands for the value for the state, and A networks stands for the advantage of each action on that state.

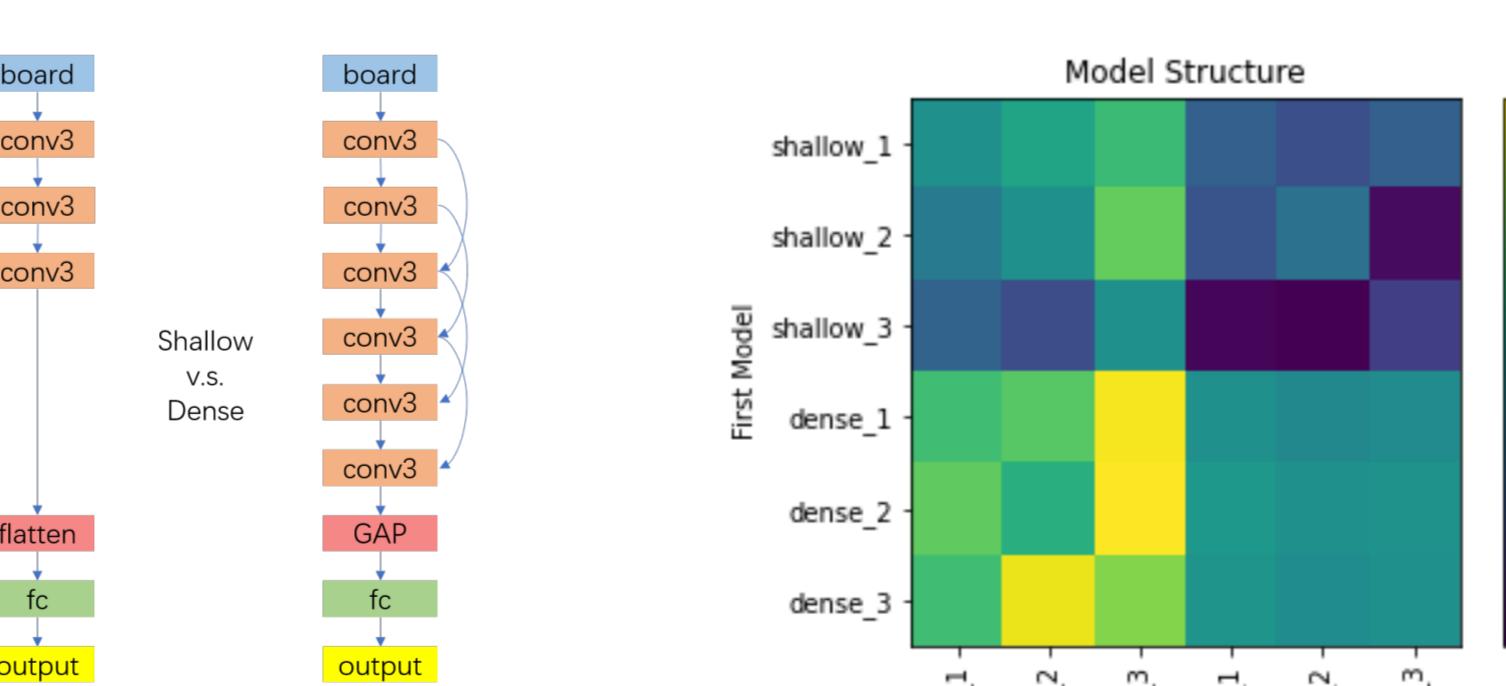
Experiment Design

Experiments were conducted on one versus one scheme for all models. For every game, we run 100 epochs and calculate the accumulated reward for both models.

Implementation trick

1. Model structure

Basically, we tried out two types of training models. The first type is shallow CNN with 3 layers, and we tried with 3 different middle channel number. Another is dense CNN with fixed channel number but has layer of 6,9,12.



From the experiment we can find out those dense CNN models outperform those shallow CNN models, while the difference among themselves are comparably smaller. The best model is dense_2 thus we use it as our model structure.

2. Board encoding

Initially, the board is encoded as 1 channel. However, we found out that split out the board into multiple channels based on the different roles may have better performance. We tried out 5 channels(1 channel for food, each snake occupies 1 channel) and 17 layers(1 channel for food, each snake occupies 4 channels, 1 channel for head, 1 channel for body, 1 channel for tail, 1 channel for the previous position of head).

Experiment Design (cont.)

	channel_1	channel_5	channel_17
channel_1	0	-36	26
channel_5	36	0	33
channel_17	26	-33	0

From the experiment we find out that encoding with 5 channels has the best performance. It seems like encoding with too many layers will degenerate the expression ability of the encoding in this algorithm.

3. Buffering strategy

We found out that the trained model has the trend to take some dangerous movements. Thus, we deliberately increase the number of training samples that the snake fails(being eliminated) in the game. We use an additional buffer to record this kind of training samples, thus we call it as double buffering.

The experiment shows that double buffering has a dominating performance over single buffering in almost all setups. Deliberately increasing the failing samples will help the model to have better handle danger position.

4. Reward function

We tried out several reward strategies. For example, One is to give -1/40 penalization for each step the snake moves, 1 reward for each food the snake eats, -5 penalization for contacting snakes' bodies. Another example is to reward only at the end of game, which gives reward of 1, 1/3, -1/3, -1 based on the ranking.

The experiment shows that the first reward function works best. It seems that giving reward continuously is better than give it at the end.

IMPALA

IMPALA is a large-scale training framework for Policy Gradient algorithms. In this framework, we will have multiple workers on CPUs to do off-policy environment exploration and one learner on GPU to do learning. The weight of the latest learner will be distributed to workers periodically.

Policy-Based algorithm outperforms Value-Based in the sense of better convergence and it does not have the problem of policy regression. However it also suffers from the issue of high variance. To reduce the variance, there are several improvements on top of the policy gradient algorithm.

$$\begin{aligned}\nabla_{\theta} J(\theta) &= \mathbb{E}_{p_{\theta}} [\nabla_{\theta} \log p_{\theta}(a|s) g_t] \\ &= \mathbb{E}_{p_{\theta}} [\nabla_{\theta} \log p_{\theta}(a|s) Q_w(s, a)] \\ &= \mathbb{E}_{p_{\theta}} [\nabla_{\theta} \log p_{\theta}(a|s) A_{w,v}(s, a)] \quad \text{Advantage Actor-Critic A2C} \\ &= \mathbb{E}_{p_{\theta}} [\nabla_{\theta} \log p_{\theta}(a|s) \delta_v] \\ &= \mathbb{E}_{p_{\theta}} [\nabla_{\theta} \log p_{\theta}(a|s) \delta_e] \quad \text{TD (\lambda) Actor-Critic} \\ &\quad \text{REINFORCE} \\ &\quad \text{Q Actor-Critic} \\ &\quad \text{TD Actor-Critic} \\ &\quad \text{TD (\lambda) Actor-Critic}\end{aligned}$$

We evaluated several of them, and the performance of the model are recorded.

Model	REINFORCE	TD Actor-Critic	TD(\lambda) Actor-Critic	V-TRACE
Max Score	1008	1046	1081	1032

References

- [1] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Ian Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.
- [2] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay, 2018.
- [3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.
- [4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.