Playing Snake with Deep Reinforcement Learning

Wong Zhao Wu
School of Computing
Singapore Polytechnic, Singapore
zhaowu.wong@gmail.com

***Abstract*—Solving classic games with Deep Reinforcement Learning (DRL) has gathered a lot of research attention. This paper proposes a Deep Q-Learning approach to solving the game of Snake up to normal human level. The focal point of this paper is to study the impact of human engineered state space towards the performance of the agent as compared to end-to-end pixel approach and the paper have found that human engineered state space could lead to a faster convergence, albeit with a sub-optimal policy ascribable to the inherent bias of the engineered features.**

I. Introduction

Contemporary contributions to Deep Reinforcement Learning often begins with training an agent in a controlled environment like the Atari 2600 games [1] to even solving the ancient game of Go [2] without any prior human knowledge given to the agent. Solving a game using reinforcement learning might sound trivial to some but unequivocally, it is a major stepping stone as the same Deep Reinforcement Learning algorithm can be translated to solve problems in variety of domain such as robotics [3] and self-driving cars [4].

Classic Reinforcement Learning problems formulates the environment as a Markov decision process (MDP) whereby current state of an environment is independent of the previous state which helps to reduce the complexity for our agent to remember and recognize the previous states and action history. Any Markov Decision Process can be solved using the Bellman Equation when the model dynamics is known to the agent (i.e. transitional and rewards probability are known for each given state and action). Traditionally, classical dynamic programming like value iteration and policy iteration are used to solve the Bellman Equation but as the input dimension and amount of observable states grows, soon it becomes inefficient as both methods require the storing of a Q-table for every given state and action pairs.

As such, Deep Reinforcement Learning is a paradigm shift to solve problems with high dimensional state and action spaces. Instead of storing a Q-table for calculation of Q-values, an Neural Network is used in place of the Q-table to approximate the Q-values given the state and action pairs. Using temporal difference learning method, Deep Q-Learning is introduced [1] and subsequently leads to other DRL methods like Duelling DQN [5], A2C [6] and TRPO [7].

This paper proposes a Deep Q-Learning based approach for the classic Snake game which originates from a 1976 arcade game Blockade and on the beloved Nokia mobile phones in 1998 [8]. The game works by letting the player to controls a snake with four distinct action available (Left, Right, Up, Down) that determines the direction which the snake will be moving. The player has to control the snake in a way it eats food and he has to avoid the frame's borders or its own body or otherwise the snake dies. For every instance where the snake eats a food, the snake body of the snake will grow in a unit.

The contributions of this paper includes a compare and contrast across different state space of human engineered features, raw pixel values and a combination of both state space to investigate its impact towards the convergence of the DQN agent.

II. Related Works

Snake is a classic video game that been with us for quite some time. Multiple attempts have been made in the literature to solve the game with or without DRL. For instance, since the game does not penalize the amount of steps it takes for the snake to get the fruit, datagenetics [9] have shown that the game is solvable by just repeating a constant route whereby the snake will not run into its own body and could travel across the full map and thus eating all the fruits along the way. Such solution is pragmatic, albeit time inefficient especially so when the size of the map is big enough that it takes $total\ sum\ grids - 1$ steps to get the fruit in the worst case scenario.

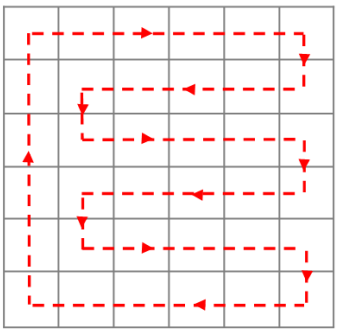

Fig. 1. Brute force solution to solving snake game. [9]

In the realm of DRL, both Alessandro et. al. [10] and Zhang et. al. [11] have attempted solving the game using DQN and Double-DQN but with different approach to the state space and rewards specifics. Alessandro et. al. has performed manual feature engineering to the game which results in a state space of just 12 binaries that encodes the information of the direction of the fruits and danger as well as where the snake is currently heading. Such feature engineering eliminates the need of a separate CNN to extract the game information and what is left is just series of Neural Network layers.

$$\begin{aligned} x(k) = [&danger\ straight,\\ &danger\ right,\\ &danger\ left,\\ &moving\ left,\\ &moving\ right,\\ &moving\ up,\\ &moving\ down,\\ &food\ left,\\ &food\ right,\\ &food\ up,\\ &food\ down]. \end{aligned}$$

Fig. 2. State space after feature engineering with prior knowledge. [10]

On the other hand, Zhang et. al. have implemented Double-DQN with CNN architecture to extract relevant features of the game from just the raw pixel values. However the best score achieved by their agent is only 10.3 points while best score in Alessandro et. al.'s work is 56.

However, such comparison of mean score across works are inaccurate as the game dynamics like the board size are different across both implementations. As such, this paper attempts to make a comparison between manual feature extraction of features based on game dynamics with prior human knowledge against an end-to-end approach with just the raw pixel values as the input and uses CNN for feature extraction. Finally, a combined approach is taken where both the raw pixel values and engineered feature space are parsed in to the model as the state space to discover what could be achieved by combining the best of both worlds.

III. METHADOLOGY

A. *Snake Environment*

The snake environments in this paper is heavily referenced from Hennie de Harder's snake repository on Github [12] that implements the game environment using turtle in python3. The game is rendered in a 20 by 20 checker box with the snake and fruit rendered with different shape and colour.

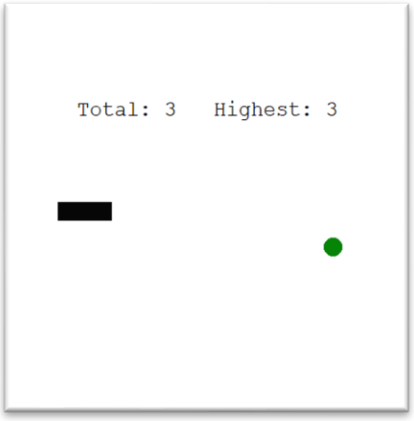

Fig. 3. Sample Snake Environment Footage.

When it comes to the exact state space parsed to our DQN agent, there are three options as discussed in earlier session. First, the state space comes in the form of manually extracted vector that encodes all necessary information for the agent to solve the game including the direction of dangers, fruits and the current direction of the snake. On the other hand, the second options would be to have a state space in a form of raw pixel values. In the original implementation, a high definition 324 pixel by 324 pixel RGB image. However, one could argue that having such a high definition state space does not aid the model in terms of actual learning in such a simple environment but only increase the time and computational cost to store all the frames in the replay memory. Thus, the image is downscaled to 80 pixel by 80 pixel RGB image as the state space for our agent. Finally, the third option combines both of the state space above and parsed to the model as a tuple instead.

Having a well defined reward that reflects the goal that we wish the agent to achieve is empirical in Reinforcement Learning. As such, when the model eats a fruit, 10 points are awarded to the agent albeit when the agent have lose the game (i.e. hit the wall or hit the body of the snake), -100 points are penalized for the model. In addition, to encourage the model heading towards the direction of the fruit, +1 point is awarded for the model in each timestep. Conversely, when the agent is heading away from the fruit, -1 point is penalized for the model. This design of this reward space is well defined as it incentivize the action of eating fruits while heavily penalize the agent when the agent has lost the game, signaling the agent to stay on the game as long as it can achieve.

Lastly, the action space available for the agent is to move top(0), left(1), down(2) and right(3). When the agent is moving on the opposite direction that the agent is heading, the action will be invalid and the agent will be moving in the original direction instead.

B. *Deep Q-Learning Algorithm*

One of the goal of this paper is to train an agent that is capable of choosing the best policy π^*that can maximize the expected discounted reward given the current state of the environment. In Q-Learning, such policy is obtained by choosing action that maximize the state-action values, Q.

$$a^* = \pi^*(s) = \text{argmax}\big(Q^{\pi}(s,a)\big)$$

However, storing of Q-values in the traditional tabular format is deemed inefficient for high dimensional state space like raw pixel values. As such, Neural Networks are used to replace the Q-tables to approximate the Q-values for any given state and action pair.

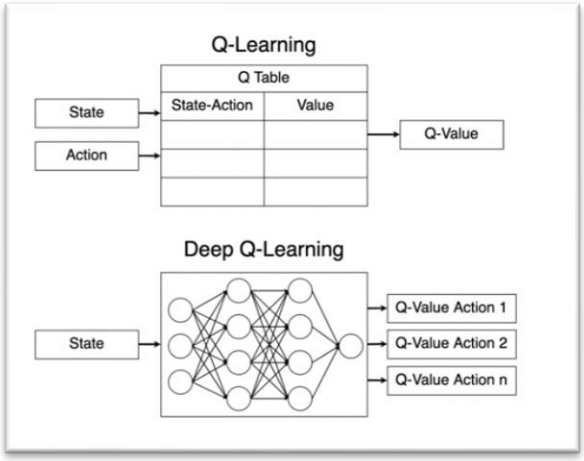

Fig. 4. Q-Learning vs Deep Q-Learning [10].

In this paper, since there are two different approach to encode the state space in format of engineered features and raw pictures, the Q-network is modified accordingly whereby convolution layers are applied for the agent with raw pixel values followed by fully connected layers for the Q-values. The model architecture will be discussed in details in the following discussion section.

As the goal of a DQN is to approximate the Q-function, training a DQN involves the same temporal difference approach whereby the DQN is update through the backpropagation of the squared distance between the expected discounted reward to the current Q-values.

$$Q(s,a) = Q(s,a) - \alpha\Big[\big(r_t + \gamma Q'(s',a';\theta') - Q(s,a;\theta)\big)^2\Big]$$

In practice, this paper have utilized a target network, $Q'(s',a')$ that is used to approximate the Q-values given the next state and action pair. The target network, parametrized by θ' is essentially the same network as DQN parametrized with θ with the target network copying the weight from the actual Q network from time steps to time steps. Freezing the target network for a while and then updating its weights with the actual Q network weights is what stabilizes the training process [1].

Another implementation details for training process is the usage of experience replay mechanism which randomly samples previous frames for training process instead of directly parsing consecutive frames. Such approach is used to break the high correlation from one state to another in the snake game environment to prevent the neural network to be highly biased and overfitted with correlated experience.

IV. Discussion

A. DQN Architecture for Different State Spaces

In this paper, there are 3 type of state space available: engineered feature vector, raw pixel values and the combination of both the state space above. As such, intuitively the DQN architecture would be slightly different across the three type of state space. For a fair comparison across the state spaces, the model architecture is kept as similar as each other as possible with only necessary modifications for each state space.

The DQN architecture for the engineered feature vectors consist of 4 fully connected layers with the first 3 hidden layers of 128 units and rectified linear unit as the activation head as better illustrated in Fig 5. For the raw pixel state space, a CNN head is required to capture the spatial features before parsing the features to the same fully connected layers as defined in Fig 3. For the CNN heads, in order to capture features with a wider receptive fields, the first convolution layer have kernel size of 8x8 pixel with depth of 64 and stride of 2. Subsequently, two additional CNN layers are appended with kernel size of 4x4 pixel with depth of 64 and 128 and a stride of 2. Finally, all the feature map extracted are flatten before parsing it to the FCs for regression of Q-values as shown in Fig 6.

Finally, the compound state space model basically combines the two model above with the raw pixel values parsed through a CNNs and the flatten feature is concatenated with the human engineered features vector before parsing it to the final FCs.

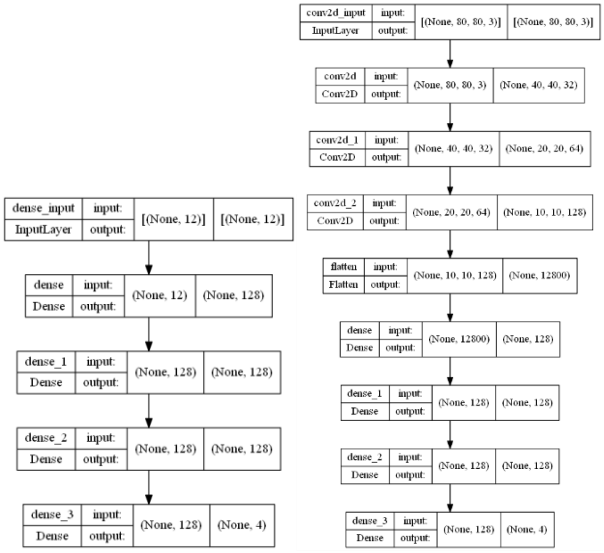

Fig. 5. Model Architecture for Engineered Feature Vector State Space.

Fig. 6. Model Architecture for Raw Pixel Value State Space.

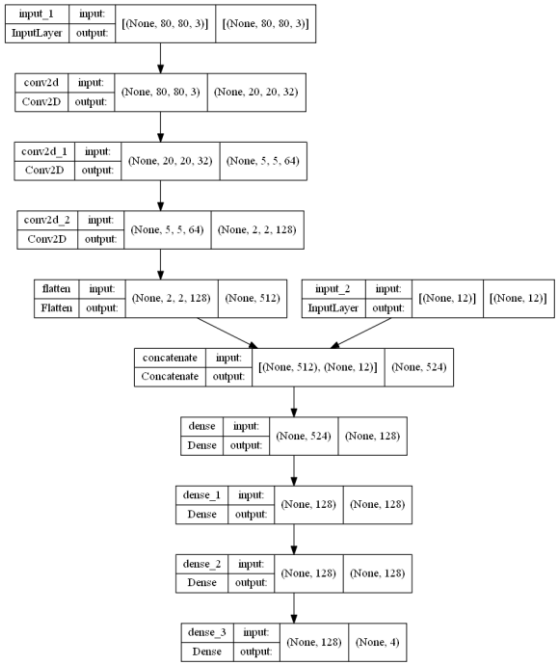

Fig. 7. Model Architecture for Compound State Space

B. Effect Modification of State Space with Model Evaluation

Out of the three state spaces, surprisingly the model with the least amount of parameter with the engineered feature have learn how to solve the game and managed to perform at human level as compared to the two other agents.

TABLE I. HIGHEST SCORE OPTAINED BY AGENTS IN 500 EPISODES

State Space	*Highest Score Obtained in 500 Episodes*
Feature Vector	45
Raw Pixel	2
Compound State Space	21

Fig 8 have shown a few screenshots of the training process for the feature vector network and clearly it has learned the techniques to find the shortest path (diagonally) to the fruits and perform tight maneuver in the corners of the screen that is deemed challenging for amateur human players. However, the technique is definitely not flawless as the agent eventually lose the game when the snake has become too long and eventually trapped itself with no other options left but to hit its own body as shown in Fig 9. Such scenario occurs as the engineered feature simple does not reveals the information of the snake body which limits the ability for the agent to "plan" a safer route before chasing for the apple. As such, a common intuition would be to find some way to reveals the actual state environment to the agent using the raw pixel values and convolution layers.

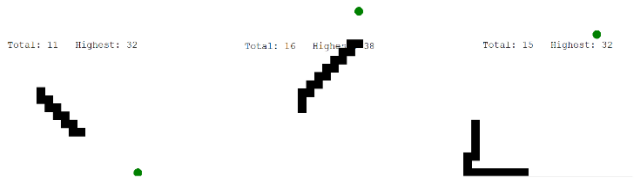

Fig. 8. Examples of Vector Space Agent solving the game with shortest distance traversing and tight corner maneuvers.

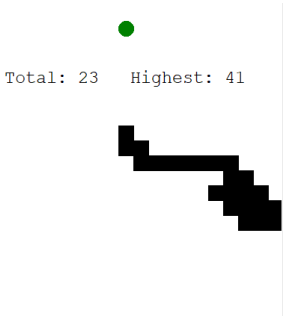

Fig. 9. Vector Space Agent got too obsessed in chasing the apple and eventually hit its own body.

Albeit in practice, both of the model that utilizes the CNN head does not learn as fast and as well as the agent with feature engineered vector space as shown in Table 1.

Such phenomenon could be explained with the increase of complexity to the optimization process for the raw pixel agent which ended up not being helpful for the convergence of the model. Besides, the hyperparameters to the convolution layers such as the kernel size and number of strides could also affect the performance of the model however, due to the interest of time and resource, I was unable to run a full sweep to find the optimal hyperparameters to improve the performance of the agent. Another final explanation to the phenomenon could be just that the training steps are insufficient for the agent to converge.

V. Conclusion

After several iteration of compare and contrast between the impact of different state spaces to the convergence of the model. The conclusion that the paper has reached is that feature engineered state space with prior knowledge could indeed simplify the task of solving the Snake game but suffers from high human bias and simply insufficient for the model to reach the best policy possible: To plan the actions in advanced and avoid running into dead end while the agent is chasing for the fruits.

It is certainly a disappointment that both the state space with raw pixel value and CNN DQN does not managed to compete against agent with engineered features and I really do not have spare time to make any further amendments. The followings are a few improvements that I wish to make if the circumstances and time allows. First, I wish to perform more fine-tuning to the hyperparameters and usage of more modern network architecture like ResNets [Deep Residual Learning for Image Recognition] for the CNN heads in order to strike a perfect balance between the model complexity and training difficulties if the computational resource allows. On top of that, I wish to explore other Q-Learning flavoured algorithms like Double-DQN which mitigates the overestimation of Q-values and SARSA which penalize the model for picking the wrong steps could potentially be useful for us to solve the game of Snake to human level standards.

.

References

[1] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," p. 9.

[2] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, Art. no. 7676, Oct. 2017, doi: 10.1038/nature24270.

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017, doi: 10.1109/MSP.2017.2743240.

[4] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A Survey of Deep Learning Applications to Autonomous Vehicle Control," *ArXiv191210773 Cs Eess Stat*, Dec. 2019, Accessed: Feb. 06, 2022. [Online]. Available: http://arxiv.org/abs/1912.10773

[5] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," *ArXiv151106581 Cs*, Apr. 2016, Accessed: Feb. 06, 2022. [Online]. Available: http://arxiv.org/abs/1511.06581

[6] V. Mnih *et al.*, "Asynchronous Methods for Deep Reinforcement Learning," *ArXiv160201783 Cs*, Jun. 2016, Accessed: Feb. 06, 2022. [Online]. Available: http://arxiv.org/abs/1602.01783

[7] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," *ArXiv150205477 Cs*, Apr. 2017, Accessed: Feb. 06, 2022. [Online]. Available: http://arxiv.org/abs/1502.05477

[8] "Snake (video game genre)," *Wikipedia*. Jan. 24, 2022. Accessed: Feb. 06, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Snake_(video_game_genre)&oldid=1067717436

[9] "Solving Snake." https://datagenetics.com/blog/april42013/index.html (accessed Feb. 06, 2022).

[10] A. Sebastianelli, M. Tipaldi, S. Ullo, and L. Glielmo, "A Deep Q-Learning based approach applied to the Snake game," May 2021. doi: 10.1109/MED51440.2021.9480232.

[11] R. Zhang and R. Cai, "Train a snake with reinforcement learning algorithms," p. 11.

[12] Hennie, *henniedeharder/snake*. 2021. Accessed: Feb. 07, 2022. [Online]. Available: https://github.com/henniedeharder/snake