Reinforcement Learning Research Project:

Deep Q-Network Agent For Snake Game

Oluwaleke Umar Yusuf

M.Sc. Robotics, Control & Smart Systems ~ Fall 2021

Introduction

Background

Reinforcement Learning (RL) involves mapping situations to actions to maximize a numerical reward signal. Within the AI context, reinforcement learning refers to goal-oriented algorithms which learn how to achieve a complex objective or maximize along a particular dimension over many steps. The learner (agent) is not told which actions to take but must discover which ones yield the most reward by trying them [1].

In RL, the agent interacts with the environment, seeking ways to maximize the reward. Thus, the agent's success is highly dependent on an accurate representation of the environment and a precise formulation of the rewards/penalties.

Traditional reinforcement learning was built upon several underlying techniques such as Monte Carlo, Temporal Difference, Q-Learning, and SARSA. Deep Reinforcement Learning (DRL) combines machine learning with the framework of RL to help software agents reach their stated goals and objectives [2].

Deep Q-Network (DQN) is the most famous DRL model which learns policies directly from high-dimensional visual inputs – thus bridging the gap between high-dimensional visual inputs and actions. DQN is based on Q-Learning, a traditional RL off-policy, temporal difference, control algorithm [3]. This algorithm implements a learned action-value function, which directly approximates the optimal action-value function, independent of the followed policy [1, p. 151].

Research Purpose & Scope

The objective of this applied research is the Python implementation of a simple DQN agent for the classic game of *Snake*. This implementation will also use the PyTorch machine learning framework and the Pygame library [4].

This research will begin by examining the theory behind the traditional Q-Learning algorithm and study how the algorithm has been integrated with modern machine learning algorithms and neural network architectures to create Deep Q-Networks.

Therefore, the implementation of the Snake agent will start by carefully examining how best to abstract the information about the agent, environment, and incentives. Such careful thought is necessary to ensure that the agent succeeds without breaking the game's rules through loopholes in the formulation.

Literature Review

One of the primary goals of artificial intelligence (AI) is to produce fully autonomous agents that interact with their environments to learn optimal behaviors, improving over time through trial and error. This section – based on [5] – briefly reviews the basics of deep reinforcement learning and deep Q-networks, providing a theoretical background for the code implementation of the *Snake* deep Q-network agent.

Deep Reinforcement Learning

In the RL set-up, an autonomous *agent*, controlled by a machine learning algorithm, observes a *state* $\boldsymbol{s}_t$ from its *environment* at timestep *t*. The agent interacts with the environment by taking an *action* $\boldsymbol{a}_t$ in state $\boldsymbol{s}_t$. When the agent takes an action, the environment and the agent transition to a new state $\boldsymbol{s}_{t+1}$ based on the current state and the chosen action. The state is a sufficient statistic of the environment and thereby comprises all the necessary information for the agent to take the best action.

The *rewards* provided by the environment determine the best sequence of actions. Every time the environment transitions to a new state, it also provides a scalar reward $\boldsymbol{r}_{t+1}$ to the agent as feedback (*Figure 1*). The agent's goal is to learn a *policy* (control strategy) π that maximizes the expected *return* (cumulative, discounted reward). Given a state, a policy returns an action to perform; an *optimal policy* is any policy that maximizes the expected return in the environment.

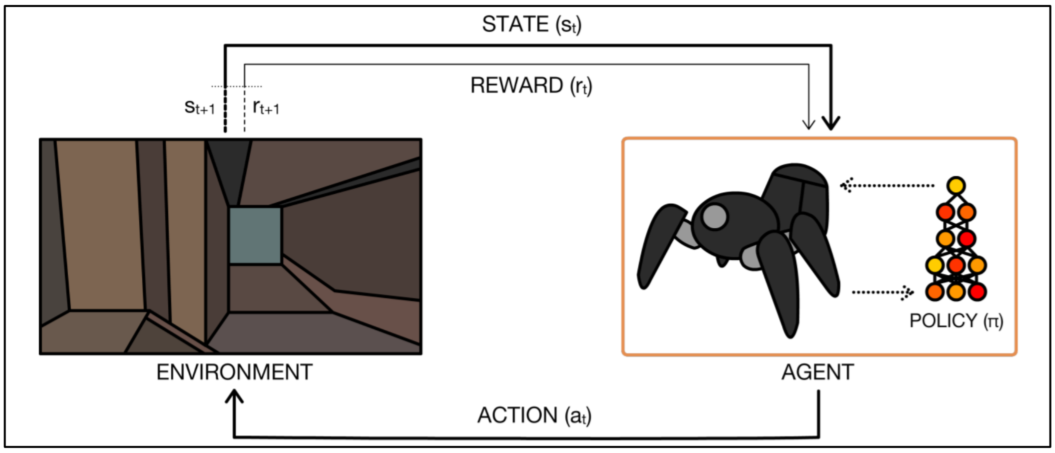

Figure 1: The perception-action-learning loop. At time t, the agent receives state $\boldsymbol{s}_t$ from the environment. The agent uses its policy to choose an action $\boldsymbol{a}_t$. Once the action is executed, the environment transitions a step, providing the next state $\boldsymbol{s}_{t+1}$ and feedback in the form of a reward $\boldsymbol{r}_{t+1}$. The agent uses knowledge of state transitions, of the form ($\boldsymbol{s}_t$, $\boldsymbol{a}_t$, $\boldsymbol{s}_{t+1}$, $\boldsymbol{r}_{t+1}$), to learn and improve its policy [5].

Deep learning – relying on the robust function approximation and representation learning properties of deep neural networks – provides new tools for RL algorithms to overcome the typical limitations of memory complexity and computational complexity. The use of deep learning algorithms within RL defined the field of Deep Reinforcement Learning (DRL), resulting in accelerated progress in RL and enabling RL to scale to problems with high-dimensional states and action spaces. DRL has found applications in domains such as robotics, video games, indoor navigation, amongst others.

Deep Q-Networks

Value function RL methods estimate the value (expected return) of being in a given state. The *state-action value* or *quality* function $Q^{\pi}(\boldsymbol{s},\boldsymbol{a})$ is the expected return when starting in a state **s**, performing an initial action **a**, and following policy **π** from the succeeding state onwards.

$$Q^{\pi}(\boldsymbol{s},\boldsymbol{a}) \ = \ \mathbb{E}[R|\boldsymbol{s},\boldsymbol{a},\pi]$$

The best policy, given $Q^{\pi}(\boldsymbol{s},\boldsymbol{a})$ can be found by choosing ***a*** greedily at every state: $argmax_a Q^{\pi}(\boldsymbol{s},\boldsymbol{a})$.

To learn Q^π, the Markov property is exploited, and the function is defined as a Bellman equation, which has the following recursive form:

$$Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q^\pi(\boldsymbol{s}_{t+1}, \pi(\boldsymbol{s}_{t+1}))]$$

This means that Q^π can be improved by *bootstrapping*, i.e., the current estimated values of Q^π can be used to improve the estimate. This is the foundation of Q-learning:

$$Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t) + \alpha\delta$$

where α is the learning rate and $\delta = Y - Q^\pi(\boldsymbol{s}_t, \boldsymbol{a}_t)$ is the temporal difference (TD) error; here, Y is a target as in a standard regression problem.

Q-learning is *off-policy*, as Q^π is updated by transitions that were not necessarily generated by the derived policy. Q-learning uses $Y = r_t + \gamma max_{\boldsymbol{a}}\, Q^\pi(\boldsymbol{s}_{t+1}, \boldsymbol{a})$, which directly approximates Q^*. To find Q^* from an arbitrary Q^π, we use generalized policy iteration, where policy iteration consists of policy evaluation and policy improvement.

Policy evaluation improves the estimate of the value function, which can be achieved by minimizing TD errors from trajectories experienced by following the policy. As the estimate improves, the policy can naturally be improved by choosing actions greedily based on the updated value function.

The DQN (*Figure 2*) addresses the fundamental instability problem of using function approximation in RL with the experience replay technique. Experience replay memory stores transitions of the form ($\mathbf{s}_t$, $\mathbf{a}_t$, $\mathbf{s}_{t+1}$, $\mathbf{r}_{t+1}$) in a cyclic buffer, enabling the RL agent to sample from and train on previously observed data offline.

Not only does this massively reduce the number of interactions needed with the environment, but batches of experience can be sampled, reducing the variance of learning updates. Furthermore, the temporal correlations that can adversely affect RL algorithms are broken by sampling uniformly from a large memory.

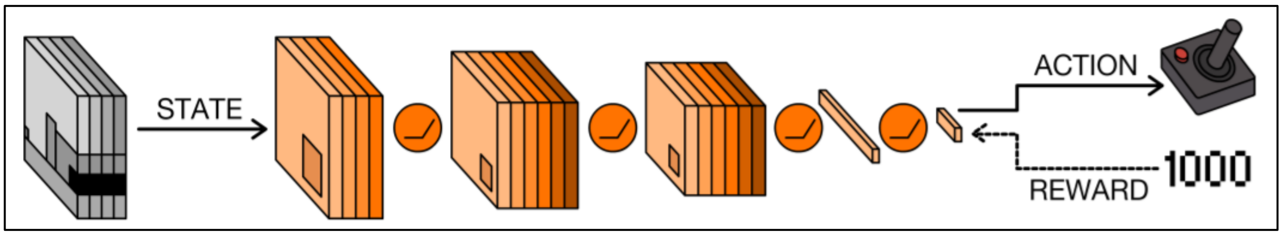

Figure 2: The deep Q-network. The network takes the state and processes it with (convolutional and) fully connected layers, with ReLU nonlinearities in between each layer. At the final layer, the network outputs a discrete action corresponding to one of the possible control inputs for the game. Given the current state and chosen action, the game returns a new score. The DQN uses the reward – the difference between the new score and the previous one – to learn from its decision. More precisely, the reward is used to update its estimate of Q, and the error between its previous estimate and its new estimate is backpropagated through the network [5].

Background

Snake (*Figure 3*) is a classic arcade video game released in the mid-1970's. The player controls a long, thin creature, resembling a snake, which moves around on a plane, picking up food while trying to avoid hitting its tail, the surrounding walls, or any obstacles within the playing area [6].

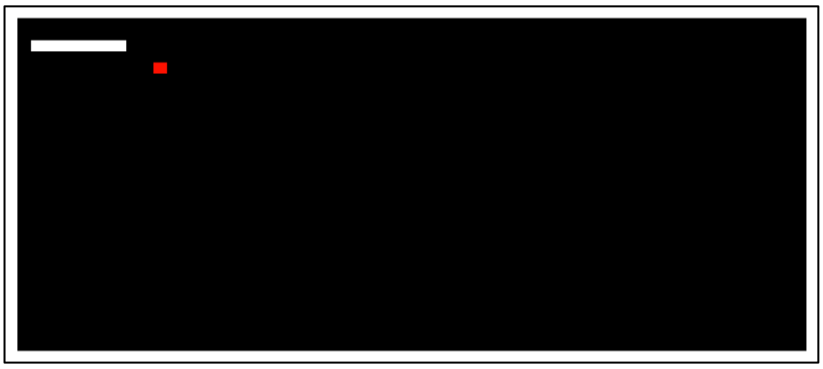

Figure 3: Classic Snake Game [7]

Each time the snake eats a piece of food, its tail grows longer – and the game speed might also increase – making the game increasingly difficult. The user controls the direction of the snake's head (up, down, left, or right), and the snake's tail follows. The player cannot stop the snake from moving while the game is in progress and cannot make the snake go in reverse [6].

Agent, Environment, Incentives & State Representation

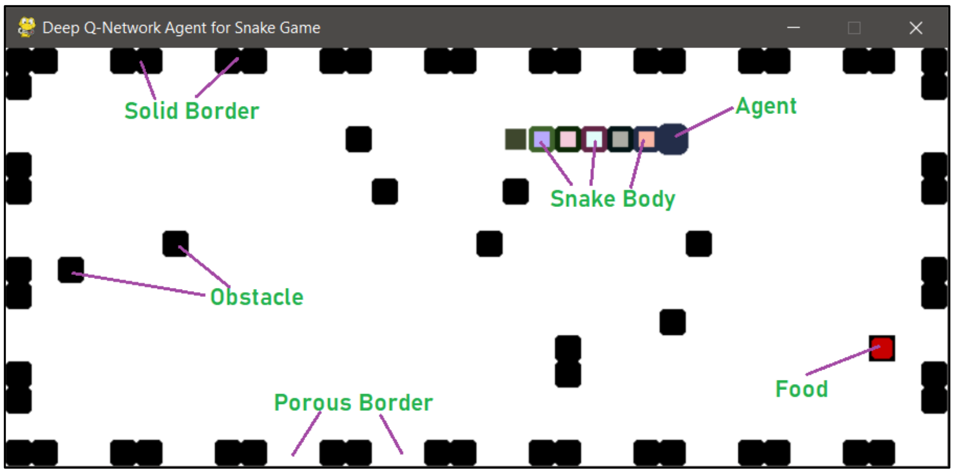

Figure 4: Deep Q-Network Agent and Environment for Snake Game

Agent

From the background given above, a reasonable mistake might be considering the snake as the agent. However, the snake's head is the agent. The interaction of the snake's head with the environment – the snake's tail, the walls (borders), obstacles, and food – determines the game's progress. Furthermore, the snake's tail always traces the path taken by the snake's head.

Environment

The environment, shown in *Figure 4*, is what the agent – snake's head – interacts with throughout the game for different rewards and penalties. The components of the environment vary and are as follows:

- **Snake's Tail**: The rest of the snake – apart from the head – is considered part of the environment, and its length increases with each food 'eaten'. Any collision between the snake's head and its tail ends the game and is penalized.

- **Walls (Borders)**: The borders of the playing area also form part of the environment and are of two types, viz:
  - **i.** *Solid Borders* – Any collision between the snake's head and a solid border ends the game and is penalized.
  - **ii.** *Porous Borders* – Any contact between the snake's head and a porous border result in the snake passing through and coming out at the opposite border with no penalty applied.

  Thus, a game can be composed of completely solid (closed) borders, completely porous (open/no) borders, and a combination of solid and porous (hybrid) borders, as seen in *Figure 5*. This serves to alter the dynamics and difficulty of the game.

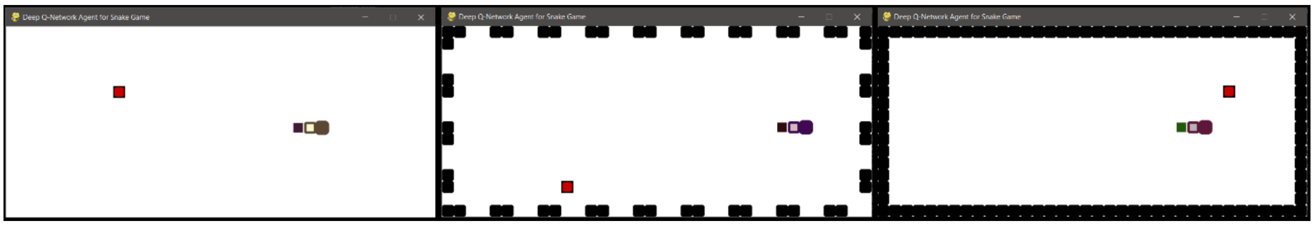
Deep Q-Network Agent for Snake Game
Deep Q-Network Agent for Snake Game
Deep Q-Network Agent for Snake Game

Figure 5: Open Borders (Left), Hybrid Borders (Middle), Closed Borders (Right)

- **Food**: This is the target of the agent and is placed randomly within the playing area. Any collision of the snake's head with the food is rewarded and increases the length of the snake's tail by one. The game's speed can also be increased; however, this only taxes the reflexes of human players. Increasing the game speed makes no difference to the AI agent and is not implemented in the code.

- **Obstacles:** These are solid borders – of varying numbers – placed randomly within the playing area during each game to make the environment more dynamic and increase the game's difficulty. As with solid borders, any collision between the snake's head and an obstacle ends the game and is penalized.

Incentives (Rewards & Penalties)

Collisions between the agent and the food are rewarded, while collisions between the agent and the snake's tail, solid borders, and obstacles are penalized. Also, lack of progress – no collision with the food, snake's tail, solid borders, and obstacles after a specified period – by the agent is penalized to prevent the agent from playing safe and simply running in loops.

Different incentives schemes can be implemented to influence the decisions made by the agent. However, an equally-weighted incentives scheme was implemented where the reward for eating the food is *+10,* and the penalties for undesirable collisions and lack of progress are *-10*. An ablation study implementing different rewards and penalties was also carried out.

State Representation

State representation refers to the simplified abstraction of all the information present in the game at any instant (in this case, frame) into a definite number of variables. The amount and type of information contained in the state representation determine the level of awareness of the agent. A delicate balance must be struck between too much information and too little information – both resulting in the agent failing to achieve the desired goal(s).

In this code implementation, the game state is abstracted into 11 binary variables – representing imminent danger (collision), agent's position (direction), and goal location. Using the game frame in *Figure 6* as an example, the game state variables are as follows:

- **Imminent Danger/Collision** – The direction is relative to the current direction of the agent. Any of these variables can be 0's or 1's at any moment.

  1. *Danger Straight* **|** ***2.*** *Danger Right* **|** ***3.*** *Danger Left*

- **Current Position/Direction** – This is relative to the environment/world, i.e., the screen or playing area. Only one of the variables can be a 1, with the others set to 0.

  4. *Direction Left* **|** ***5.*** *Direction Right* **|** ***6.*** *Direction Up* **|** ***7.*** *Direction Down*

- **Goal/Food Location Heuristic** – This gives the 'Manhattan direction' of the food relative to the environment/world. This 'Manhattan direction' heuristic is a simplified version of the Manhattan distance, and only one or two of these variables can be 1's.

  8. *Food Left* **|** ***9.*** *Food Right* **|** ***10.*** *Food Up* **|** ***11.*** *Food Down*

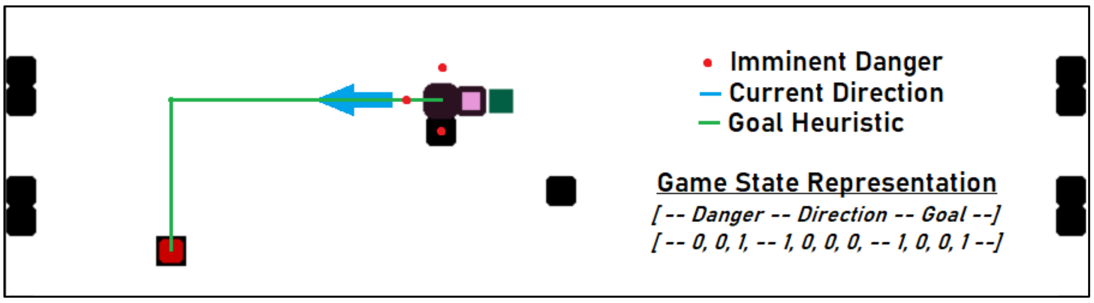

Figure 6: Game State Representation Example

Code Implementation in Python, PyTorch & Pygame

The code implementation in this work builds upon a similar open-source project [8] by Patrick Loeber [9]. The Python code implementation is split into four modules:

_dqnAINetwork.py

This helper module contains the PyTorch implementation of the neural network (NN) models and architectures – simple (*11 x 256 x 3*) and complex (*11 x 256 x 512 x 3*). The module also contains the Q-Learning algorithm for calculating the loss across game states and training the agent. The NN hyperparameters include *Adam* optimizer, *ReLU* activation function, *Mean Squared Error* loss function.

_dqnSnakeEnvironment.py

This helper module contains the Pygame implementation of the game environment. The placement of borders, obstacles, food and the snake across the playing area is done within this module. This module outputs information needed to abstract the game state and inputs information to determine the next move of the agent and updates the user interface.

_dqnAIAgent.py

This helper module implements the snake game agent by building upon and connecting the ***_dqnAINetwork.py*** and ***_dqnSnakeEnvironment.py*** helper modules. It takes in and abstracts information from ***_dqnSnakeEnvironment.py*** to determine the game state variables and passes the state variables to ***_dqnAINetwork.py*** for training per frame. It also queries ***_dqnAINetwork.py*** for information about the agent's next move and passes that back to ***_dqnSnakeEnvironment.py***. The ***_dqnAIAgent.py*** module has a maximum memory of 100000 frames, i.e., it can only remember the past 100000 state variables for training the network at the end of each game.

dqnSnakeMain.py

This is the main module for running the Snake Game Deep Q-Network code implementation. It contains three functions – which depend on the ***_dqnSnakeEnvironment.py*** and ***_dqnAIAgent.py*** helper modules – viz:

- *AITrainer()*: Creates an environment from given border and obstacles specifications, trains an agent for a specified number of games, and saves the trained agent model to disk.
- *AITester()*: Creates an environment from given border and obstacles specifications, loads a trained agent model from disk, and tests the agent for a specified number of games.
- *userMode()*: Creates an environment from given border and obstacles specifications and sets up a game where the user's keyboard input determines the snake's movement.

Results & Discussion

Training and Testing Results

Several deep Q-network agents were trained on different combinations of environment and incentives – borders, obstacles, rewards, and penalties. Furthermore, agents trained on one combination of environment and incentives were tested on other combinations of environment and incentives to investigate how much the agent's training generalizes when faced with novel situations.

The agents were trained for 500 games, and each test also lasted for 500 games. The training and testing ablation studies conducted are enumerated in *Table 1* below.

Table 1: Training and Testing Ablation Studies Conducted on DQN Agents

	Agent, Environment & Incentives	**Testing/Training**	**Max. Score (500 games)**	**Mean Score (500 games)**
1	**Simple Neural Network Closed Borders Equal Rewards and Penalties *0 Obstacles**	*-- Training --*	64	-
		0 Obstacles	64	28.54
		10 Obstacles	61	23.36
		20 Obstacles	58	18.72
	Simple Neural Network Closed Borders Equal Rewards and Penalties *10 Obstacles	*-- Training --*	50	-
		0 Obstacles	51	18.69
		10 Obstacles	46	11.58
		20 Obstacles	41	7.25
	Simple Neural Network Closed Borders Equal Rewards and Penalties *20 Obstacles	*-- Training --*	41	-
		0 Obstacles	57	18.04
		10 Obstacles	43	10.98
		20 Obstacles	34	7.64
2	**Simple Neural Network Hybrid Borders Equal Rewards and Penalties *0 Obstacles**	*-- Training --*	74	-
		0 Obstacles	76	22.86
		10 Obstacles	72	17.19
		20 Obstacles	55	13.26
	Simple Neural Network Hybrid Borders Equal Rewards and Penalties *10 Obstacles	*-- Training --*	48	-
		0 Obstacles	**78**	20.47
		10 Obstacles	46	12.34
		20 Obstacles	39	8.29
	Simple Neural Network Hybrid Borders Equal Rewards and Penalties *20 Obstacles	*-- Training --*	35	-
		0 Obstacles	69	16.10
		10 Obstacles	51	8.76
		20 Obstacles	42	5.92
3	**Simple Neural Network Hybrid Borders Different Rewards and Penalties** *[collision=-10, food=+30, no progress=-10]*	*-- Training --*	30	-
		20 Obstacles	17	2.91
4	**Testing < Closed > Borders Agent in Environment with < Hybrid > Borders**	*0 Obstacles*	56	23.40
		20 Obstacles	51	16.44
	Testing < Hybrid > Borders Agent in Environment with < Closed > Borders	*0 Obstacles*	69	28.37
		20 Obstacles	46	15.28
5	**Complex Neural Network** **[11 x 256 x 512 x 3]* **Closed Borders Equal Rewards and Penalties 0 Obstacles**	*-- Training --*	59	-
		0 Obstacles	64	**28.95**
		10 Obstacles	60	19.70
		20 Obstacles	47	12.50

Discussion

Table 1 shows the mean and maximum scores of different DQN *Snake* agents across 500 games both in training and testing. It should be noted that any score above 50 is impressive as the length of the snake at that point makes navigation very hard. The gameplay shown in *Figure 7* achieved a final score of 47, and it can be seen that the agent almost trapped itself around its tail twice.

The overall maximum score (78) was achieved by an agent trained on hybrid borders with obstacles and tested in an environment with no obstacles. It is also interesting to note that the testing score of 78 was higher than the training score of 48. The same can be observed across the entire table – agents performed as well or better during testing as in training.

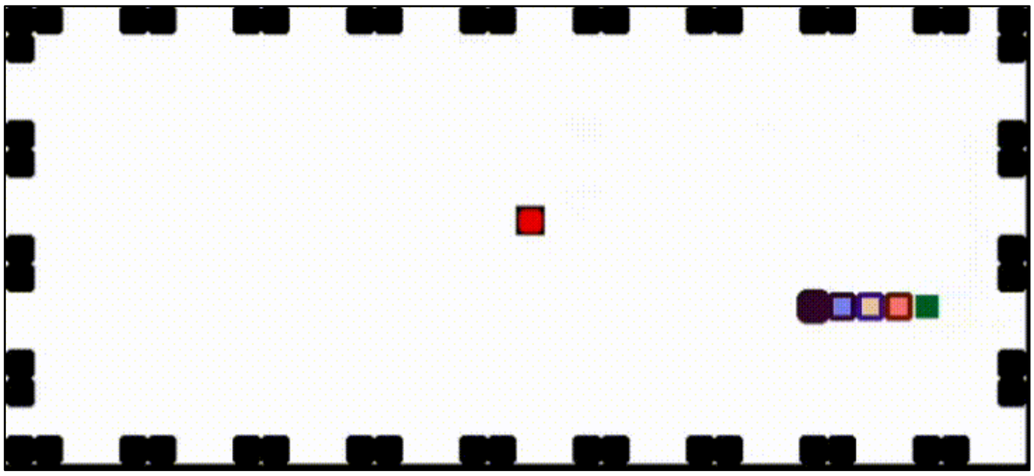

Figure 7: Deep Q-Network Agent Trained and Tested on Hybrid Borders.

Figure 8 compares the training and testing scores for a sample agent. While the final scores per game are noisy, the mean training score gradually improves while the mean testing score stays constant and close to the training value.

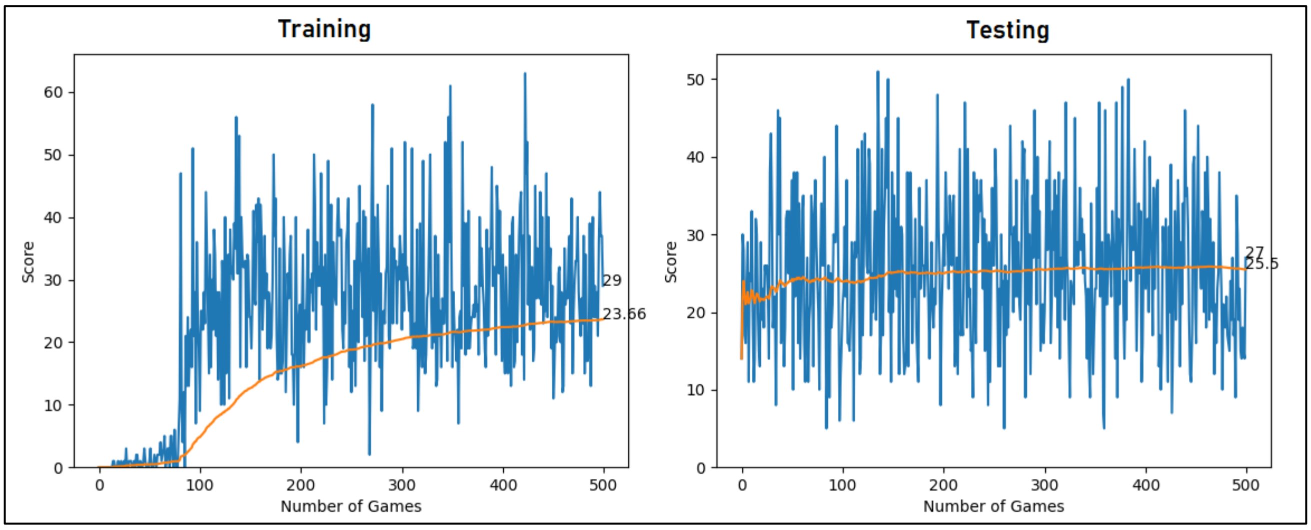

Figure 8: Mean (Orange) and Final (Blue) Training and Testing Scores for a DQN Snake Game Agent

The rest of this section shall briefly go over the effect of different borders, obstacles, neural networks, and incentives on the training and testing results of the DQN *Snake* agents, viz:

<u>Effect of Borders:</u>
Agents trained on hybrid borders had higher maximum scores than agents trained on closed borders. This is understandable as the porous borders give the agent a way out of tight spots that will otherwise result in a collision.

However, the hybrid-borders agents had lower mean scores than closed-borders agents. During training and testing, it was observed that hybrid-borders agents get confused and looped from one porous border to the other in a straight line.

Effect of Obstacles:

It is interesting to note that agents trained without obstacles performed way better – higher maximum and mean scores – than agents trained with obstacles during testing, regardless of the border type. Intuitively, one would expect the opposite, that agents trained on obstacles will find a testing environment without obstacles much easier to navigate. Instead, the agents would often get confused with the snake chasing its tail.

From a machine learning perspective, agents trained without obstacles generalized better to testing environments with obstacles while agents trained on obstacles overfitted to that environment, despite changing the positions of the obstacles every game to prevent the agent from memorizing the obstacles.

Effect of Neural Network Architectures:

Most of the testing and training were done using a relatively simple neural architecture (*11 x 256 x 3*). An agent trained with a more complex architecture (*11 x 256 x 512 x 3*) in an environment with closed borders and no obstacles showed no overall improvement in performance over the agent trained with the simple architecture in the same environment.

When an even more complex model (*11 x 256 x 512 x 256 x 3*) was tried, the agent did not learn at all. It is well-known that the higher the number of parameters in a neural network, the more the amount of training data required. It appears that the snake game agent with 11 state variables cannot provide enough training data across 500 games for complex network architectures.

Effect of Incentives (Rewards & Penalties):

Most of the training was done with equally-weighted incentives of ±10. However, to correct agents' confusion in environments with hybrid borders and obstacles, an agent was trained with a higher reward of +30 for eating the food. The agent – trained in an environment with hybrid borders and 20 obstacles per game – performed worse (mean score of 2.91) than the agent trained with equally-weighted incentives in the same environment (mean score of 5.92).

Furthermore, the result was even worse when an agent was trained with incentives of +50 for food, -10 for collisions, and -30 for lack of progress. Note that the lack of progress encompasses the snake looping from border to border and chasing its tail.

Effect of Cross-Domain Training and Testing:

To test the amount of cross-domain knowledge gained by the agents, the agent trained on closed borders with no obstacles was tested in an environment with hybrid borders and 0/20 obstacles. The reverse was also done; the agent trained on hybrid borders with no obstacles was tested in an environment with closed borders and 0/20 obstacles.

As noted earlier on the effect of borders on the agents' performances, the hybrid-borders agent had better maximum scores (0 obstacles/20 obstacles) of [69/46 vs. 64/58] while the closed-borders agent had better mean scores (0 obstacles/20 obstacles) of [23.40/16.44 vs. 22.86/13.26]. The hybrid-borders agents are more explorative and risk-taking, while the closed-borders agents are more exploitative and conservative.

Conclusion

The Deep Q-Network Agent for *Snake* Game code implementation, training, and testing was an educative experience. Creating the environment and its components is more straightforward than figuring out the metaheuristics of best game state representations, incentives schemes, and neural network architectures.

Furthermore, all three metaheuristics are linked, and the suboptimality of any will affect the agent's training. The game state variables serve as the inputs to the neural network. Two sets of game state variables before and after an action/move are combined with the incentives (rewards and penalties) to calculate the network's loss and train the agent/model.

Thus, a game state representation without sufficient information or irrelevant information will not give good loss values when combined with the incentives. A suboptimal incentive scheme might interact negatively with the state representation and neural network and induce undesirable behavior in the agent. A badly-chosen neural network architecture might either overfit or underfit and negatively impact the agent's training.

Thus, the three metaheuristics of game state representation, incentives scheme, and neural network architectures are much more critical and harder to figure out than the game environment and its components. Further investigation into these three metaheuristics will help mitigate the issues that plague the snake game agents developed so far. These issues are:

i. The testing scores per game vary by significant amounts. A better outcome will be for the score per game to be within the 50th percentile of the training maximum. This will indicate that the agent has learned not to make 'silly' mistakes.

ii. The agent tends to trap itself around its tail, usually when the tail gets long. The current game state representation does not account for the position of the snake's head relative to its tail. Adding variables representing the proximity of the snake to its tail can help avoid self-entrapment.

iii. The agent tends to chase its tail, especially in the presence of obstacles. The addition of state variables representing the proximity – not just imminent collisions – of the tail and obstacles can help prevent this issue.

iv. The agent tends to loop from border to border in the presence of porous borders. A better incentives scheme that better penalizes such behavior can be adopted. Furthermore, a state variable that monitors this behavior can improve the network's training or even forcibly shift the agent in a different direction.

v. The neural network hyperparameters – learning rate, loss function, activation function, etc. – used for training are generic ones commonly used. A better combination of hyperparameters more suitable for reinforcement learning can yield better training results.

In conclusion, the Q-learning algorithm is a simple yet powerful reinforcement learning method that is easy to implement and fast to train and yields agents and models with outstanding performance when combined with deep, machine learning.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: A Bradford Book, 2018.

[2] N. Chris, ‘A Beginner’s Guide to Deep Reinforcement Learning’, *Pathmind*. Available: http://wiki.pathmind.com/deep-reinforcement-learning. [Accessed: 12-Oct-2021].

[3] C. J. C. H. Watkins, ‘Learning from Delayed Rewards’, PhD Thesis, King’s College, Oxford, 1989.

[4] ‘About - Pygame’. Available: https://www.pygame.org/wiki/about. [Accessed: 13-Oct-2021].

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

[6] ‘Snake (video game)’, *Encyclopedia Gamia Archive Wiki*. Available: https://gamia-archive.fandom.com/wiki/Snake_(video_game). [Accessed: 11-Nov-2021].

[7] ‘Pixilart - Snake Game (Gif Test)’, *Pixilart*. Available: https://www.pixilart.com/art/snake-game-gif-test-16c3630a9147a08. [Accessed: 11-Nov-2021].

[8] L. Patrick, *Teach AI To Play Snake! Reinforcement Learning With PyTorch and Pygame*. https://github.com/python-engineer/snake-ai-pytorch, 2021.

[9] L. Patrick, ‘Teach AI To Play Snake - Practical Reinforcement Learning With PyTorch And Pygame’, *Python Engineer*. Available: https://python-engineer.com/posts/teach-ai-snake-reinforcement-learning. [Accessed: 12-Oct-2021].