Assessing the Impact of State-Space Complexity on an Image based DQN via the Game of Snake

David J. Richter
Chonnam National University
Gwangju, South Korea
0000-0001-5413-6710

Kyungbaek Kim
Chonnam National University
Gwangju, South Korea
0000-0001-9985-3051

***Abstract*—Not too long ago Reinforcement Learning, one of the three fundamental approaches to Machine- and Deep Learning, was mostly incapable of handling complex and difficult tasks. This, however, has since changed and Reinforcement Learning agents, most of them utilizing Deep Reinforcement Learning, have mastered ever more difficult problems. One of the key aspects to consider when designing Reinforcement Learning systems is the state observation, which is what the agent "sees" of the task environment and is also used as the input to the Deep Neural Network on top. In this work we will explore how the state-space complexity of the image based observations impacts the training and final performance of the DQN agents, and how having data rich states compares to creating states that are more data efficient. In this work we will experiment with different possible state-space representations, all of which contain a full observation of the current game state, and inspect how they will influence the agents performance and/or how the agent has to account for said states, in order for them to perform well.**

***Index Terms*—Reinforcement Learning, Deep Learning, DQN, State Representation, Snake, Video Games**

I. Introduction

Reinforcement Learning (RL) has gained popularity after showing a lot of promise in a multitude of different tasks in recent years, most notably due to the advances in Deep Reinforcement Learning (DRL), which has shown to have better-than-human capabilities. Reinforcement Learning learns in a trial and error type fashion, where an agent learns online on an environment. The agent sees said environment through a state representation, interacts with it via actions and receives rewards for them. A Deep Neural Network is then used as a function approximator to estimate how much future reward the agent can expect per state-action pair, which form the optimal policy by picking the best state-action pair at each time-step. This Deep Reinforcement Learning methodology has been refined and popularized by Mnih et al. [1], who designed and introduced the Deep Q-Networks (DQN). Since then, RL has picked up a lot of steam and has been applied to a large number of different fields, which include, but are not limited to: robotics [2], flight [3], autonomous driving [4], imaging [5], gaming [1], among many more.

One field that has seen a lot of attention, ever since the beginning of Deep RL, are games. Video Games inherently offer themselves up to be used with RL since they already, by nature, posses all the prerequisites that RL requires, those being the state representation (screen rendering), the action space (controller inputs) and a reward function (highscores). In this work we will also be experimenting with a game, the game of snake, which has also already seen some work in the past [6]–[9]. But in this paper, instead of trying to beat any highscores or trying to fine-tune hyperparameters, we will look at how the state-space itself will impact the agent. We will look at different state-representations, using the game of snake as our test environment, to see how they impact the agents. We compared 17 different states with different complexity (all full observations of the game) and compared their performance. These findings will hopefully aid future works when creating image based state-spaces, even beyond snake. The contributions of this paper include an implementation of the game of snake and incorporating it into the DRL workflow (Open Source), a comparison of 17 different state representations (both from previous work as well as our own), and an analysis of the results of those different state-representations.

II. Background

A. Related Work

Reinforcement Learning, boasts a long history [10]. But its recent renaissance can be attributed in large parts to the work by Mnih et al. [1], who developed a widely-applicable and highly-capable method, accelerated by Deep Learning (DL). This Deep Q-Network approach, which utilizes Neural Networks (NN) was the first of its kind to do so in a way that allowed it to solve a multitude of different and complex tasks, showcasing a new ceiling in potential to the field of RL. RL has even managed to beat the world champion of Go [11].

Reinforcement Learning has seen application in the fields of robotics [2], autonomous driving [4], as well as gaming [12] (among many more). Snake is not without prior research either. Ma et al. [6] use a heavily reduced state representation to learn the game of snake via the Q-Learning and SARSA algorithms. As both these algorithms are not utilizing Deep Learning methods, there is a need to reduce state this much. In this work we will not consider this state design, as it would not work with a CNN model and therefore would conflict with the other state-space designs that we are looking at.

In [7] Wei et al. are using DQN with prioritized replay memory to train their agent. The state is made up of 4

sequential RGB images (all reprocessed screenshots of the game) to present the current state as well as motion.

Almalki et al. also ran experiments with RL in snake in [8] using SARSA, but do not describe the states, actions or rewards used and also do not present quantitative results.

Tushar et al. use a heavily pre-processed state representation in their paper [9], reducing the RGB images used in [7] to simple grey-scale images (still multiple images in sequential order), while also getting rid of the prioritized replay memory in the process. Results were good, but many different parameters were changed when compared to Wang et al.'s work, so it it hard to draw conclusions simply based on the difference in state-space-representations.

B. Reinforcement Learning

Reinforcement Learning works rather different than other DL methods. In RL there is no dataset and there are no labels/instructions. In RL, agents learn "on-the-fly", by interacting directly with the environment in which they are supposed to solve the given task. The agent starts of with no knowledge of the task, the environment, the actions it can control or what is considered to be good or bad. This means that the agent will first act completely at random (random actions at any given state), and over time learn what actions are considered good/bad at what state via the reward function, maximizing the possible future rewards over time, by picking the best possible action for the given state at every time-step. The next few subsections will explain this in a bit more detail.

C. Environment

The environment is the world/scenario that the agent can interact with/exist in. In the case of this paper the environment would be the snake game. The environment will present the agent with a state and a reward at every time-step, while also receiving and then applying actions sent from the agent. One full run through the environment (here one full game of snake) is considered an episode. Agents usually needs thousands if not hundreds of thousands of episodes to master the task.

a) States: States are the representation of the current condition of the environment that the agent receives. This is the only way the agent can "see" what is going on (which is why they are so important) and also used to learn as the input to the agent's NN. In the case of snake, the state depicts the current information about the game, e.g. where the snake is located, where the fruits are, etc., often via a screenshot.

b) Actions: The actions are the control commands sent form the agent to the environment. They are generated & chosen by the NN and impact the environment upon reception. In the snake use-case, the actions would most be the game controls, those being up, down, left and right.

c) Rewards: Rewards are the mechanism that actually allow learning. States and actions alone would allow the agent to "play", but only with random controls. Rewards tell the agent whether it is currently doing well, or not. Rewards are usually given per action per state and need to be designed by the person designing the experiments. In the case of snake, rewards could be: positive (reward) for eating a fruit, negative (punishment) for running into the wall.

D. Agent

While the environment offers the problem and the potential to solve it, the agent is the "brain". The agent is the one that learns, via the NN it contains. Here it could be compared to a human learning and playing the game.

a) Networks: In DQN, a single agent has two networks. A target network and a policy network. The target network is only updated every n episodes, but when it gets updated it just copies the policy network. The policy network is the "normal" network that get constantly updated (every time-step). Actions are taken from the less volitile target network however, as that improves stability.

b) Q-Values: Q-Values are what the networks are trying to predict. They represent the expected future rewards (the amount of rewards the agent can expect if taking action a in state s at time-step t as well as the best action for all future steps). Agents are trying to maximize Q-Values.

c) Replay Memory: Usually, Neural Networks are given batches from huge datasets to train on, to prevent overfitting. In RL, however, at each time-step, there only exists a single state-action-reward pairing to train on, which is far from ideal. To compensate, DQN uses Replay Memory, a "dynamic dataset' of a fixed size. The newest state-action-reward pairs get saved in the replay memory and once it has reached its capacity, the oldest ones get deleted again. This helps to stabilize training. Agents can batch from the replay memory during training.

d) Exploration & Exploitation: To ensure that the agent explores the environment well for possible strategies but later focuses on and optimizes a good one, the exploration and exploitation trade-off needs to be considered (also called ϵ-greedy). What this means is that at first the agent acts at random (100% random: $\epsilon = 1$), in order to explore the environment for all possible policies in search for the best possible one, but later abandons that randomness to focus on a good policy and exploit that policy for the rest of the training. For that the ϵ value will decay over time and reach a minimum value (1% leftover randomness: e.g. $\epsilon = 0.01$).

E. Snake

The objective of the snake game is rather simple, the snake grows longer the more fruits it eats. Eating fruits earns the player points, forcing them to keep eating fruits in the hunt for a highscore, but in turn also consciously complicating the game, as touching the body of the snake with the snake's head will kill the snake (as will touching the walls).

III. Methodology

This paper will focus mostly on different state-representations. For this purpose, a number of different state-representations have been added and implemented [13] to our testbed. This will hopefully help researchers when designing state-spaces in the future.

A. Snake Game

The game of snake was written in compliance with previous work. The size of the field was 12x12, the snake started with a length of 3 at a random location with a distance of 3 cells from the wall (unless stated otherwise (see Section IV-B)) while facing a random direction. There was always only a single fruit on the field at a time. Fruit would always respawn in a new random location, as did the snake (random location & facing random direction). Colliding with the walls and the snake's body would trigger a "Game Over" and reset the episode.

B. Reinforcement Learning

In this section we will explain the RL setup used in this work.

1) Actions: The action-space we chose for this set of experiments was rather simple and straight-forward. Just like in the video-game, the agent has 4 possible control choices, those being Up, Down, Left and Right.

[Up, Right, Down, Left]

Actions are discrete, as they need to be, by nature of the DQN algorithm, but in this case this is just fine. Continuous actions would make no difference here.

2) States: With states being the main point of interest of this work, multiple different state-space representation methods have been tested here.

a) RGB Screenshot: The first approach is a simple screenshot of the game, that was only slightly modified to allow the agent to see movement even in a single frame. To achieve that the head of the snake was painted in a color different from the color of the body. The screenshot is downscaled to simplify it for the NN. A simplified example of this state can be seen in Figure 1.

b) Minimal: The most condensed and minimal representation of the game of snake is the minimal state representation, that we designed. Here the game state is converted pixel by pixel, meaning that if the game of snake is running on a 12x12 grid, then the state space will also only represent the grid with 12x12 datapoints. Each color is assigned to a single integer value, with the head having it's own, representing movement. This approach can be seen in Table I.

c) RGB Sequential: The approach taken in [7] was implemented also. The agent receive the most recent 4 screenshots from the game (unmodified in appearance, besides preprocessing) as a single state, so that the agent can identify motion by seeing the transitions from one image to the next. This will result in a state as it can be seen in Figure 2(a).

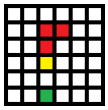
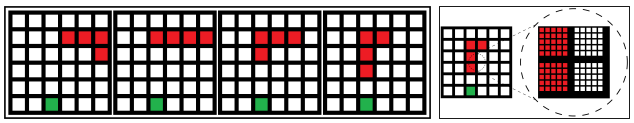


Fig. 1: Sample of a non-deep screenshot. Note: Here the field shown is of size 6x6, whereas the actual field will be of size 12x12. The snake's body is red, the head (to portrait motion) is yellow and the fruit is green.

TABLE I: Visualization of the minimal tabular state (would be 12x12 in experiments). The snake's head is represented by a 4, the body by 1s, and the fruit by a 7.

0	0	0	0	0	0
0	0	1	1	0	0
0	0	1	0	0	0
0	0	4	0	0	0
0	0	0	0	0	0
0	0	7	0	0	0



(a) Frame Stacking **(b)** True Size

Fig. 2: (a): A Representation of what the 4 sequential screenshot state looks like. Here the motion is shown via multiple images (not the head). (b): This figure showcases the screenshot states true size. One logical pixel does is made up of multiple image pixels.

3) : Noteworthy Additional Information

a) Pixels: As mentioned before, the screenshot state methods are not pixel-for-pixel, meaning that one pixel of the in-game snake logic is represented by multiple image pixels (see Figure 2(b)). This bloats the size of the state, unclear from previous work whether this is beneficiary, detrimental or plainly irrelevant.

b) RGB: Additionally, each in game pixel is also represented by not just one of these multi-pixel pairs, but by 3, one for each color channel (see Figure 3), further bloating the size and further cementing our curiosity.

c) Black and White Sequential: Lastly, we will also implement a state representation very similar to the one used in [9]. The state representation in this paper is very similar to the one in [7], but instead of having 3 channels (RGB), they are condensed to a black and white image of a single channel per image. The state is however still of depth 4 (4 sequential images) in order to maintain motion information.

4) Rewards (constant): The reward function is very similar to the one suggested in [7], with some heuristical changes. The reward function can be seen in Equations 1, 2 and 4.

$$\begin{aligned} r_{conditional} &= 25 \mid True\ if\ Fruit\ eaten \\ r_{conditional} &= -50 \mid True\ if\ Dead \end{aligned} \tag{1}$$

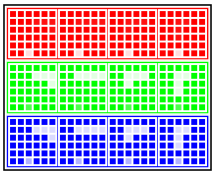


Fig. 3: Visualization of RGB sequential states. The RGB channels would also be split like this for any other state that uses RGB (e.g. Figure 1).

TABLE II: CNN Architecture

Layer	Info
Input	(State)
Conv2D	(f=16,k=(3,3),relu)
Dropout	(d=0.2)
Conv2D	(f=16,k=(3,3),relu)
Dropout	(d=0.2)
Flatten	(-)
Dense	(n=64,a=linear)
Dense	(n=4,a=linear)
loss=mse, optimizer=adam, lr=0.001	

This conditional ($r_{conditional}$) award is only applied when the snake fulfills the conditions listed. A high reward is given when the agent eats a fruit and a even higher punishment is handed out when it dies.

$$d(h, f) = \sqrt{(x_h - x_f)^2 + (y_h - y_f)^2} \tag{2}$$

This is the Euclidean distance between the head (h) of the snake and the fruit (f) using their (x, y) coordinates. It calculates how far the snake it from the fruit at every timestep.

$$r_{distance} = (d(h_{t-1}, f_{t-1}) - d(h_t, f_t)) \ / \ 10 \tag{3}$$

The distance reward ($r_{distance}$) measures the difference between the previous and the current distance (using the Euclidean distance from Equation 2), giving positive reward if the snake is moving closer to the fruit and negative reward if it is moving away from it.

$$r_{total} = r_{conditional} + r_{distance} \tag{4}$$

The rewards are then combined by simply adding them and handed to the DQN.

5) DQN: Q-Learning uses the Bellman equation (see Equation 5) to calculate the Q-Values, in DQN we still use Bellman to create labels for the Memory Replay, but we use the Neural Networks as Q-Value function approximators.

$$Q(s_t, a_t) = Q(s_t, a_t) + lr * (r + \gamma * Q(s_{t+1}, a_{t+1})) \tag{5}$$

Here $Q(s_t, a_t)$ represents the Q-Value for state s_t and action a_t at timestep t. lr stands for the learning rate, r represents the reward and $gamma$ is the discount factor.

We have opted to model our DQN similarly to what it was looked like in previous papers [7], [9], while trying to keep it simple yet capable enough to handle the task. The NN we opted for is a CNN (Convolutional Neural Network) with both regular and priority replay memory implementations. Epsilon Decay and Policy & Target Networks are implemented in accordance with the regular DQN approach.

a) Neural Network (constant): The NN architecture is one of a rather simple CNN, with 2 convolutional layers, corresponding dropout layers and 2 dense layers post flattening (see Listing II). Of course, with this the number of trainable values will vary from state-space to state-space, as different states are of different dimensionality which will lead to more depth layers after each convolutional layer.

Algorithm 1: The Snake DQN Game & Learning Loop

Init Replay Memory D with max. size N
if *using Priority Replay* **then**
  Init Priority Replay Memory $\hat{D}$ with max. size $\hat{N}$
end
Init Policy Network Q with random values θ
Init Target Network $\hat{Q}$ with random values $\hat{\theta} = \theta$
Init Snake Environment E
for *episode e = 1, M* **do**
  Reset E and *get* initial State $s_{t=0}$
  for *timestep t = 1, T* **do**
    if *Explore with probability* ϵ **then**
      Select Action a_t from $\hat{Q}$ via s_t
    else
      Select Action a_t at random
    end
    Send a to E and *apply* it.
    Receive Reward r and New State s_{t+1} from E
    if *using Deep States* **then**
      Append State to Deep State List s_{t+1} of size L
    end
    Append r_t, s_t, s_{t+1}, a_t to D (and $\hat{D}$)
    Train Q with batch of r, s, s, a pairs from D (and $\hat{D}$)
    Set Current State to be New State: $s_t = s_{t+1}$
    Decay Epsilon Value ϵ with Decay Rate d: $\epsilon = \epsilon * d$
  end
  if *Every C Episodes* **then**
    Set $\hat{Q}$ to Q: $\hat{\theta} = \theta$
  end
end

b) Replay Memory (constant): We implemented priority replay memory, to follow the approach of [7], where the "normal" replay memory buffer holds 50,000 former timesteps and the priority replay holds 25,000. Timesteps that have received rewards of 0.005 and higher are being fed to the priority buffer, with all other being sent to the regular one. Once the normal buffer reaches 1,000 and the priority one reaches 330. training will begin. The split of priority and normal experience to be sampled into the batch is 50/50.

c) Parameters (varying): The parameters used for the experiments can be seen in Table III.

d) Loop (varying): The training Loop that drives the RL process can be seen in Algorithm 1. This Loop is mostly constant across all experiments, and the main idea and workflow behind it is completely constant. State Depth (Sequential Images for Motion) are what varies the loop, but only slightly.

IV. Results

In 3 different experiment setups we will compare the resize factor (and via that the image size) as the observation space and how different sizes impact performance, then we will use

TABLE III: Parameters used

Parameter	Value
Discount Factor	0.99
Replay Memory	Section III-B5b
Minibatch Size	64
Update Target Net Rate	5
Episodes	15,000
Max Steps per Episode	150
Network Architecture	Table II
Input Layer Size / States	Section IV
Action Space	4
Reward	Equation 4
Replay Memory	Section III-B5b

Parameter	Value
Epsilon Start	1
Epsilon Decay Rate	0.9995
Episode before Decay Start	31
Epsilon Minimum Value	0.001
Board Size	12x12
Board Border	+1
Snake Starting Length	3
Snake Starting Position	Section III-A
Snake Starting Direction	Section III-A
Number Fruits	1
Fruits Position	Random

one of the top performing resize factors and use it to test for state depth, again to see which one does best. Then, lastly, we will compare the approach we have proposed (minimal size), with the black and white approach by [9] and the RGB deep image approach by [7].

A. Comparing Image Size

In order to run these images we used the screenshot of our implementation of snake as the input to the DQN agent. The screenshot was then resized by a factor as listed below. For this set of runs, we kept the state depth (number of sequential images) to 1, so in order to portrait motion we have colored the snake head in yellow, as mentioned before. When looking at the results of the experiments (see Figure 4) one can see that resize factor 9 did the best (with smaller images still generally doing better than large ones). By nature of down-sampling images, it is possible that the smaller the images get (e.g. here resize factor 15) get modified to a point where some information gets lost or distorted. Overall one can, however, see that the really large images are performing worse by comparison and smaller images do better. Also, one can see how much slower the bigger images are in real time (training hours, see Figure 5). Resize factors 3, 5, 11 and 13 were also tested but not plotted for better visibility. 13 was similarly strong in performance to 9 and 3, 5 and 11 were comparable with 1.

B. Comparing State Depth

Next we compared the impact of state depth (see Figure 2(a)). For this we will use an image resize factor of 9 as it was among the top performing ones in the previous experiment. Here we have (except for depth 1) removed the yellow color from the snake head and kept it all red, in order for the motion to be portrayed only through depth (as is [7]). As all images are RGB images and therefore inherently have a depth of 3 already (RGB channels) the depth will be multiplied by 3 to get the actual depth of the input (e.g. depth 4 means 4 RGB images, so 12 total layers). Also, usually agents were spawned at a random location at a distance of 3 fields from the walls, this was changed after a depth of 4, so that the agent would never be able run into the wall before the snake is fully built. When looking at the results (see Figure 4) we can see that all agents perform at a level that is almost identical, state depth seems to not have too much of an impact on the performance of the agent. The difference in training time is also less significant. Depths 2 and 4 were also tested and performed very similar to the other factors.

TABLE IV: Resize factors and their corresponding image size. RGB images have a third dimension of size 3.

Resize Factor	1	3	5	7
Image Size	196x196	66x66	40x40	28x28
Resize Factor	9	11	13	15
Image Size	22x22	18x18	16x16	14x14

C. Special Cases

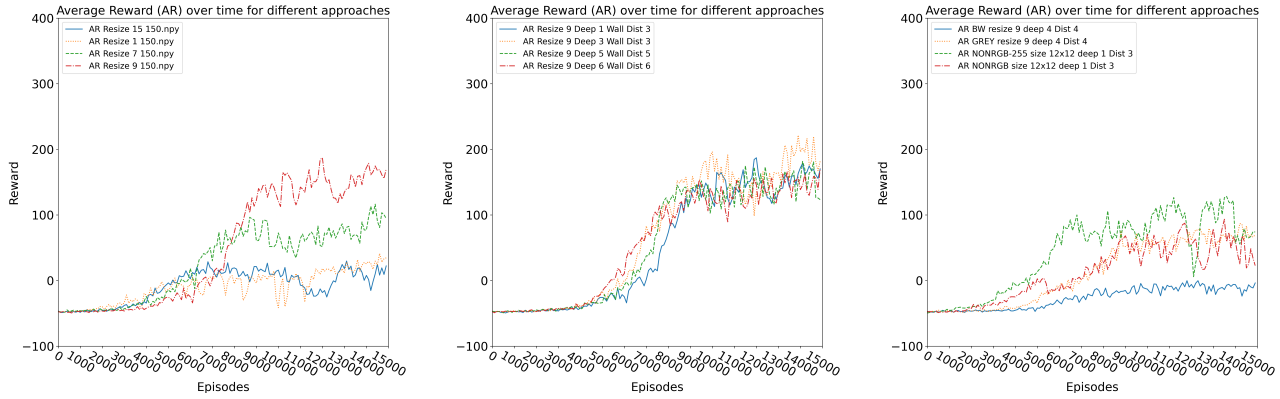
In this section we will discuss and compare the performance of the minimal state that we suggest, the black and white approach by [9], a greyscale agent, and the 4 deep and resize factor 9 approach mentioned above (similar to [7]) with the parameters that worked better according to our findings. When looking at the graph (see Figure 4) we can see that the RGB agent that did really well in the afore mentioned runs also performed best here. Our simplistic approach (see Table I) did learn a working strategy but the performance was outclassed by the RGB agent (it did learn the fastest early on, before somewhat collapsed). The greyscale agent (similar to the black and white approach in [9], but instead of only 0s and 1s, here values are either 0 or 255) did perform on a similar level to the simplified state agent. The pure black and white agent did barely learn in our setup. With that, we decided to also adjust our simplified agent to have values that range from 0 to 255 (background = 0, body = 80, head = 160, fruit = 240). As can be seen, this agent did a lot better, learning a good policy and doing so faster than all other agents, suggesting that this scaling is beneficial).

V. Discussion

As can be seen, different complexities of states have large impact on the performance of the agents, even though all state representations are full observations of the current state of the game. As such it can be seen that smaller image size states did outperform more complex observations, generally speaking, and they took less time. In terms of depth, no big difference was observed. The only state representation that did worse than the others was the one without state depth (but smaller depths trained in less time). The special cases introduced did not perform better than the best depth and resized approaches, but the minimized agent with larger scaling (as mentioned above) did perform comparatively and learned quite a bit faster.

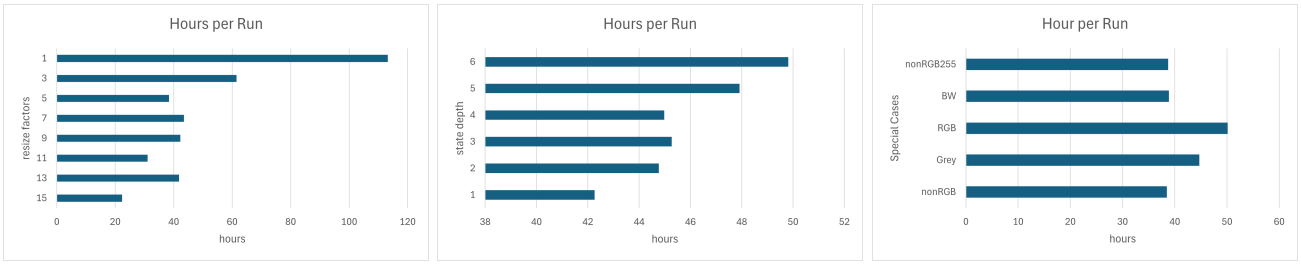
VI. Conclusion

In this work we have compared 17 different complexities to portrait the state space of the game of snake, all of which are holding a full representation of the game state at each timestep. As such we can guarantee that not the amount of necessary game data is impacting the results. In our comparison we have found that image size had a large impact on performance, with agents that had a large resize factor (smaller images) performing better. Image depth on the other hand had little to no impact. The set of special cases that minimize the state space did produce well performing agents, but did not manage to outperform the best agent of the resized runs. Scaling values from 0 to 255 did better for both the black and white agent and the minimized one, compared to scaling from 0 to 1 (or 7 respectively). These findings will hopefully help future work in picking suitable and well performing state representations for their DQN agents, when using images as states.



(a) Image Size **(b)** Image Depth **(c)** Special Cases

Fig. 4: The rewards over time for different sizes (a), different depths (b) and special cases (c).



(a) Image Size **(b)** Depth **(c)** Special Cases

Fig. 5: Training time to reach 15,000 episodes. All special cases were run with more debugging enabled, so times between graphs are not fully comparable, but within graphs they all ran with the same settings.

ACKNOWLEDGEMENTS

This work was supported by Innovative Human Resource Development for Local Intellectualization program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT)(IITP-2024-RS-2022-00156287, 34%). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2023-RS-2023-00256629, 33%) grant funded by the Korea government(MSIT). This work was supported by Korea Institute of Planning and Evaluation for Technology in Food, Agriculture and Forestry(IPET) through the Agriculture and Food Convergence Technologies Program for Research Manpower development, funded by Ministry of Agriculture, Food and Rural Affairs(MAFRA)(project no. RS-2024-00397026, 33%).

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] F. Niroui, K. Zhang, Z. Kashino, and G. Nejat, "Deep reinforcement learning robot for search and rescue applications: Exploration in unknown cluttered environments," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 610–617, 2019.

[3] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for uav attitude control," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 2, pp. 1–21, 2019.

[4] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[5] I. Kostrikov, D. Yarats, and R. Fergus, "Image augmentation is all you need: Regularizing deep reinforcement learning from pixels," *arXiv preprint arXiv:2004.13649*, 2020.

[6] B. Ma, M. Tang, and J. Zhang, "Exploration of reinforcement learning to snake," 2016.

[7] Z. Wei, D. Wang, M. Zhang, A.-H. Tan, C. Miao, and Y. Zhou, "Autonomous agents in snake game via deep reinforcement learning," in *2018 IEEE International Conference on Agents (ICA)*. IEEE, 2018, pp. 20–25.

[8] A. J. Almalki and P. Wocjan, "Exploration of reinforcement learning to play snake game," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2019, pp. 377–381.

[9] M. R. R. Tushar and S. Siddique, "A memory efficient deep reinforcement learning approach for snake game autonomous agents," in *2022 IEEE 16th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE, 2022, pp. 1–6.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[12] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A survey of deep reinforcement learning in video games," *arXiv preprint arXiv:1912.10944*, 2019.

[13] D. J. Richter, "pythonsnake," https://github.com/JDatPNW/pythonSnake, 2023.