Deep Reinforcement Learning with Varying Levels of Observability in *Snake*

Robert Chandler & Joshua Day

Contents

1. Introduction

In real-world reinforcement learning applications for Markov processes, it is typically impossible to have total access to the underlying state of the environment; instead, the agent must utilize whatever observations of the true state it can make, making the problem a partially-observable Markov decision process (POMDP). The accuracy of these observations may vary widely depending on the environment, the methods used to obtain the data, and what fiscal and computational resources are available. Obtaining better observations almost always implies a higher monetary cost (greater quantities and/or types of sensors with greater accuracy levels) and a higher level of complexity (the observation space may quickly grow too large to solve analytically, training times will increase as the dimensionality of features grows). In light of this, our objective is to design a virtual environment which lends itself to multiple levels of observability, employ several different observation space models in this environment, train deep learning/neural RL algorithms on each space, compare the results, and comment on insights gleaned.

We use the classic video game *Snake* as our experimental environment. It strikes the desired balance between simplicity in its implementation and adaptability to multiple observation spaces, and it is one of the most universally understood video games to date, being well-loved since its first appearance in 1976 [1]. The game has applications to various real-world scenarios in which an agent is tasked with collecting objects or covering certain areas while avoiding collisions, including automated cleaning tasks (lawn-mowing, vacuuming, etc.), collecting cargo or debris with a vehicle, or even herding animals.

The viability of an observation is somewhat bound by what its possible actions are. For example, if the snake can only turn left, right, or stay straight, then it it must know which direction it is currently moving in order to make a proper decision. We allow the agent to move the snake absolutely in one of the four cardinal directions rather than relative to its current direction, so this additional information is not necessary to encode in the observation space.

We propose the following observation models in order of complexity (implementations are discussed in Section 3.2.1):

1. The agent only observes whether there is a dangerous tile in one of the eight grid spaces surrounding its head and whether the current food item is above, below, left, and/or right of the head. We refer to this as the *directional* observation space.
2. The agent observes the relative position of the current food item as a 2D coordinate as well as the distance of the closest obstacle in each of the four cardinal directions from the snake's head. We refer to this as the *positional* observation space.
3. The agent observes the state of the entire grid. In this case, the environment could be considered a fully-observable MDP. We discuss it further in Section 3.2.1, but we encode this space two different ways:
    1. The observation is a 2D grid of integers where each value represents a different state under the square. We call this the *numeric grid* observation space.
    2. The observation is a stack of 2D grids of binary elements where each grid represents a different state and the binary state indicates whether the square at that location is the state corresponding to the grid or not. We call this the *discrete grid* observation space.

2. Background and Related Work

2.1. Game Mechanics

The objective of *Snake* is to maneuver a snake character around a grid of discrete playable positions collecting food items while avoiding collisions with the walls or the snake itself. Each time a food item is obtained, the length of the snake increases, making self-collision harder to avoid. When the snake collides with a wall or itself, the player loses, and when the snake occupies every playable space of the board, the game is won.

Figure 1: Our implementation of *Snake*, rendered in the terminal

The game can be considered an episodic Markov process where either a winning or losing state marks the termination of each episode. While we can formulate this problem as a Markov chain, it is not mandatory that we do so, and in our solutions for the grid-based observation spaces, we employ a recurrent neural network (RNN) to train our policy, which incorporates time history into the learned policy, and therefore does not satisfy the Markov property.

2.2. Theoretical Analysis

Snake has been studied rigorously in the field of graph theory [2], and we show that some of these concepts directly apply to the context of reinforcement learning. We desire to investigate the effect of the observation space on the resulting learned policy. We consider a case where each observation is the position of the snake's head in the grid and there are four possible actions corresponding to moving the head in one of the cardinal directions.

Claim: If the $m \times n$ grid has a Hamiltonian cycle, then there exists a deterministic, stationary policy π that can win the snake game.

Proof: Assume an $m \times n$ grid has a Hamiltonian cycle $(v_1, v_2, ..., v_{m \cdot n})$ (a closed path in a graph which visits each vertex exactly once). From this cycle we can define a function $\alpha : \mathcal{S} \to \mathcal{A}$ which takes a state and returns the action in the grid that sends each vertex to the next vertex in the cycle, i.e.

$$P(v_i, \alpha(v_i)) = \begin{cases} v_{i+1} \text{ if } i < mn \\ v_1 \text{ if } i = mn \end{cases}$$

where P is the transition function of the MDP. We can define our stationary, deterministic policy as $\pi(v_i) = \alpha(v_i)$.

To show that this policy will ultimately win the Snake game, consider the start of the game. If our agent (the size of a single cell) follows the of the Hamiltonian cycle, we will visit each of the mn cells in the grid before repeating, and thus we must have visited at least one food cell on the grid after completing a full cycle. Furthermore, as long as the game remains unfinished, i.e. the length of the snake l is less than mn, there will always be $mn - l$ empty cells in front of the snake following

the Hamiltonian cycle. Thus following this policy, the snake will not collide with a wall or it's tail, and continue gaining food every loop until the game is won.

□

With this strategy, we can also determine a worst-case upper bound on the number of time steps required to win the game. In the worst case, we would traverse each of the mn cells and grow by a single cell each loop through the cycle. Since the games ends when the snake consists of mn cells, we can guarantee to beat the game in time $t \leq (mn)^2$.

While this result is not universally applicable as it relies on the existence of a Hamiltonian cycle through the graph, it does have a few interesting implications. Since finding a Hamiltonian cycle is an NP-Complete problem, this provides some intuition towards the difficulty and nature of the problem for which we desire our agent to solve.

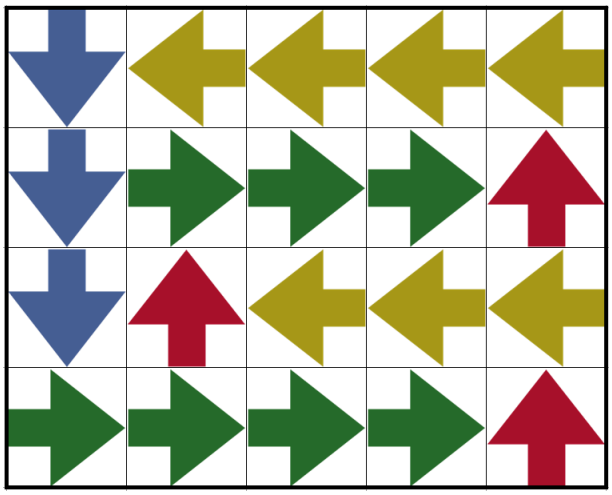

Figure 2: A grid with an even number of rows showing the induced stationary, deterministic policy from a Hamiltonian cycle.

Furthermore, from the work of D. Graafsma [2], we see that if we have either an even number of rows or columns, then we can construct a Hamiltonian circuit in a rectangular grid. One such example is provided in Figure 2. Finally, it is also noted that "Although odd-sized grid graphs are non-Hamiltonian, we can construct a cycle that contains all vertices except for one. In fact, we can find two of these cycles that differ by exactly one vertex." This means that we can still construct a winning policy in the case where both the number of rows and columns are odd, but it is no longer deterministic or stationary unless we augment the observation space with new information.

2.3. Design of the Observation Space

In a 2024 paper from J. J. Garau-Luis, R. K. Goel, E. Crawley, and C. Wu, observation space design is studied from an algorithmic point of view, which is much deeper than the scope of this report, but their motivation is the same as ours [3]. They state:

> One of these key RL components is the observation space. The choice of observation space has a direct impact on the performance of the agent (Reda et al., 2020; Singh et al., 2020; Kim & Ha, 2021) and it often consists of hand-crafted sets of features. The literature around concrete applications demonstrates that in many cases there is no convergence on which feature sets to use. For example, recent studies on RL for Traffic Signal Control (TSC) (Wei et al., 2021; Higuera et al., 2019; Zhou et al., 2019) propose more than 10 different feature sets to solve the same problem (Noaeen et al., 2022). Another common approach to observation space design

is using feature sets of high dimensions (e.g., adding features for completion, for symmetry, etc). While adding extra features may not confer any additional advantage (Guyon & Elisseeff, 2003), it may not be desirable from a practical perspective either, as every extra feature may entail additional data that must be collected, which may be more intrusive for users or incur substantial hardware costs.

They identify the same issues that we do in observation space design: there are an effectively infinite number of ways to formulate the design space for a problem, and the advantages that introducing a more complex space will bring are impossible to know up-front. Indeed, the only guarantee of introducing a space with higher dimensionality is the increase in cost and complexity.

2.4. DQN Learning

We train all four of our environments via the Deep Q-Network (DQN) algorithm for consistency. This algorithm, which has shown proven success on Atari games, uses a deep neural network to approximate the Q function. The formulation of this network will necessarily be unique to each environment, but the DQN loss function for each is identical:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right],$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a called the *behavior distribution* [4].

We technically are using the Double DQN (DDQN) form of this algorithm, which builds on the DQN algorithm by using two separate Q-value networks to avoid maximization bias [5].

We use the same number of maximum overall steps used for training for each algorithm to keep a consistent baseline for comparison as well.

3. Methodology

3.1. Overview of Software Libraries

Choosing a strong software stack to support our implementation is critical given the code-heavy nature of this problem. We chose Python as our programming language since it is by far the most popular choice for RL applications and AI in general, largely in part due to the strong ecosystem of supporting libraries, some of which we mention here.

3.1.1. Gymnasium

To model *Snake* in an RL-friendly manner, we created a custom environment using Gymnasium, an open-source Python project maintained by the Farama Foundation (formerly known as Gym and maintained by OpenAI), which provides a standard API for developing reinforcement learning environments [6].

While there were pre-existing open-source implementations of *Snake* that we could have modified to hook into a specific RL library, the Gym/Gymnasium standard is the most prevalent in the reinforcement learning community, and using a common standard helps to ensure that we do not become trapped into a certain ecosystem, giving us the option to switch RL libraries with much less friction than if we had manually interfaced a specific RL library with a preexisting implementation of the environment.

We also could have taken the approach of hooking an existing implementation into a custom Gymnasium environment, or even modifying an existing Gymnasium environment for the game, but given that our main objective is to have a highly customizable environment, we decided it best

to create our own. This also provided a much deeper understanding as to what is going on under the hood when environment interactions occur.

3.1.2. TorchRL

After defining our environment, the other critical component required is a library which provides a framework for designing algorithms that can be used to train our agent to interact with the environment. We chose TorchRL, the official RL component of the massively popular PyTorch library [7]. While most of the popular RL libraries utilize PyTorch models to implement their algorithms, we were drawn to TorchRL for several reasons. First of all, it is an official component of the PyTorch ecosystem, so it has first-class support and integration with all of the existing PyTorch tooling. It also introduces the powerful `TensorDict` data structure, which makes working with rollouts very intuitive, and provides first-class support for multi-agent reinforcement learning (MARL), which was originally part of our scope (see Section 5.2).

3.2. Implementation Details

3.2.1. Snake Environments

We implement four separate environment classes: one for each observation space model mentioned in Section 1. We begin by building a `SnakeBase` class which inherits from the abstract `Env` class and defines all of the common code which each of our four environments will inherit.

Every Gymnasium environment has three main requirements:

1. An initialization method `__init__` which constructs all the state necessary for a new environment and defines the `action_space` and `observation_space` properties using the `gymnasium.spaces` module. We utilize a `spaces.Discrete` class for our `action_space` with one discrete action for each of the four cardinal directions.
2. A `reset` method, which can be used to reset the state of the environment after an episode terminates.
3. A `step` method, which takes an `action` and returns:
    1. A new `observation` which belongs to the `observation_space` defined.
    2. A `reward` for the action. Our reward function is simply $+1$ if a food item was obtained, -5 if a collision occurred, and 0 otherwise.
    3. Whether the process was `terminated` or `truncated` as a result of the action.
    4. Any additional `info` desired. In our case, we provide the current `snake_length`, which is our main performance metric, the `current_direction` the snake's head is pointing, the L1 `distance` to the nearest food, and whether the agent has `won` the game.

Our `SnakeBase` class implements all of the game state and logic, including helper methods for rendering the current board state as a string, which is essential for visualizing algorithm performance and learned behaviors. On this note, many environments utilize a graphical user interface (GUI) to visualize the states of the environment. Ours generates a string that can be printed in any terminal emulator and we also provide a module for generating animated versions for playback via the tool `asciinema`, which can record and play back output live in the terminal and can optionally render the animations as GIFs.

To keep our scope manageable, we limit the size of the playable board to 5-by-5 spaces (7-by-7 when including a layer of walls surrounding the border).

The only part of `SnakeBase` which is left un-implemented is the `observation_space` property and the `get_obs` method. Each environment overrides these appropriately.

3.2.1.1. Directional Observations

The `SnakeDirectional` environment observes the directions of "dangerous" spaces surrounding it and the current direction of the food:

```
self.observation_space = spaces.Dict(
    {
        "danger": spaces.MultiBinary(8),
        "food": spaces.MultiBinary(4),
    }
)
```

We define eight binary values representing whether or not the snake tail or a wall is in each of the eight spaces surrounding the snake's head. Additionally, we define four binary values indicating whether the current food item is above, below, left, and/or right of the snake's head. This is a fairly limited observation of the entire board, but it provides the essential information: where the reward states are and where the terminal states are.

3.2.1.2. Positional Observations

The `SnakePositional` environment observes the relative position and distance of the current food and any the nearest "dangerous" spaces, respectively:

```
self.observation_space = spaces.Dict(
    {
        # Distance from the head in each direction
        "nearest_danger": spaces.Box(
            low=1,
            high=max_distance_1d,
            shape=[4],
            dtype=self.dtype,
        ),
        # The position of the food relative to the head
        "food": spaces.Box(
            low=-max_distance_1d,
            high=max_distance_1d,
            shape=[2],
            dtype=self.dtype,
        ),
    }
)
```

Where `max_distance_1d = 5`: the maximum straight-line distance from the head from the snake's head to an opposing wall.

The `Box` class describes a numerical space (can be discrete or continuous) which is bound between two values. It can have arbitrary shape, and our `nearest_danger` one has shape `(4,)` for each of the cardinal directions, while the `food` one has shape `(2,)` as it represents a relative position in 2D space.

The `self.dtype` parameter allows us to use a specific data type for our numeric values, but this is typically just a `float32` since this is what many of the PyTorch modules expect as the input type.

3.2.1.3. Grid Observations

The `SnakeGrid` environment observes the full state of the game, including the walls, yielding a 7-by-7 array of integer values enumerating the possible states: `EMPTY, WALL, HEAD, TAIL, FOOD`:

```
self.observation_space = spaces.Box(
    low=0,
    high=4,
    shape=[self.full_size, self.full_size],
    dtype=self.dtype,
)
```

3.2.1.4. Discrete Grid Observations

Lastly, the `SnakeGridDiscrete` class observes the full state of the game board just like the `SnakeGrid`, but it encodes its values as discrete members of an enumeration. This is technically more accurate than the former implementation, which assigns numerical values to certain states, when really there is no inherent meaning in the numerical value other than that it uniquely identifies a state.

```
num_discrete_values_per_location = np.full_like(
    self.all_indices, len(States)
)

self.observation_space = MultiDiscrete(
    nvec=num_discrete_values_per_location,
    dtype=self.dtype,  # type: ignore
)
```

We first define `num_discrete_values_per_location` as a 7-by-7 array full of the value 5, indicating that each space in the grid can take one of five unique types/states. This environment looks practically identical to the previous one when comparing them at the Gymnasium level; the difference comes into play when they are actually utilized: TorchRL will keep the former as a 7-by-7 array of numerical values, while it will convert the latter into a stack of one-hot encoded arrays, yielding observations of shape `(5, 5, 7)`.

3.2.2. Environment Transforms

Each environment requires a series of transformations to convert the observations into forms that TorchRL modules are willing to accept. For example, we used the `spaces.Dict` space type to define the `observation_space` in each of the non-grid environments so that we could easily tell what the values corresponded to. However, the MLP expects to take in a tensor of values rather than a labeled dictionary of values, so we use the `CatTensors` transform to concatenate the values together into one tensor input. The `SnakeGrid` environment returns tensor observations of shape `(7, 7)`, but the convolutional neural network (CNN) that it processes it expects it to be of shape `(1, 7, 7)`, so we apply the `UnsqueezeTransform` to add the extra dimension. Lastly, the `SnakeGridDiscrete` environment outputs tensors of shape `(7, 7, 5)`, but the CNN expects `(5, 7, 7)`, so we use the `PermuteTransform` to alter the order of the dimensions. All of our environments use the `StepCounter` transform to track how many steps they have taken during each episode and limit the number of steps that an agent can take in a single episode, truncating the episode early if necessary. Lastly, the grid environments receive a special `InitTracker` and `TensorDictPrimer` transform, which are both related to tracking/labeling the hidden/recurrent states used by the RNN module.

3.2.3. TorchRL Models and Training Algorithms

Although the general outline for training is the same for all environments, each one has its own training script, giving us the freedom to tweak individual runs without affecting a modular piece of code that all other environments use. There are several command-line interface (CLI) modules as well, which allow the user to alter hyperparameters via the terminal when running the training code.

The difference in observation space can sometimes require different training implementations. For example, while we use a recurrent neural network (RNN) to learn the policy in both of the grid-based environments, we rely on a multi-layer perceptron (MLP) in the "directional" and "positional" environments. The RNN utilizes convolutional models to take advantage of the positional information available in the grid-based observations, and it also incorporates time-history information. In theory, this brings it closer to the level of reasoning that a typical human would use while playing the game.

Differences in approximation models aside, the general outline for each training script looks something like this:

Initialize environment and apply necessary transformations to observation outputs
Initialize deterministic policy module
Initialize ε-greedy policy
Initialize replay-buffer data structure for storing values
Initialize loss function, optimizer (Adam), and updater modules
$S \leftarrow$ number of overall steps to take
$b \leftarrow$ batch size to rollout during each iteration of the outer loop
$n_{\text{opt}} \leftarrow$ number of iterations to perform in inner optimization loop
$b_{\text{opt}} \leftarrow$ number of iterations to perform in inner optimization loop
$s \leftarrow$ skip this many iterations of the outer loop between policy evaluations
$i \leftarrow 0$
while number of total steps sampled $< S$ **do**
    Sample a rollout of data with b steps and append it to the buffer
    for $j = \left[1, ..., n_{\text{opt}}\right]$ **do**
        Sample b_{opt} steps from the buffer
        Calculate loss of sample
        Compute gradient of loss using `backward()` method
        Step the optimizer to update parameters
        Step the updater to update the target network
    if $\text{remainder}(i, s) = 0$ **then**
        Evaluate policy deterministically (without exploration) and log results

4. Results

The primary metric that we are concerned with comparing for these algorithms is the maximum snake length (score) that was achieved by a trained agent across all evaluation points. Additionally, we could use maximum number of steps in an episode as a secondary measure of performance, but this should be used with caution since the agent frequently gets stuck in endless deterministic action loops, arbitrarily increasing the step count until the episode is truncated.

4.1. Maximum Scores

As shown in Figure 3, we obtained the maximum scores using the Directional and Positional observation spaces, which were both able to beat the game with a maximum score of 25, while the Discrete Grid had a max score of 22, followed by the regular Grid at 17. For context, we the researchers were able to achieve a maximum score of 23 after several rounds of gameplay.

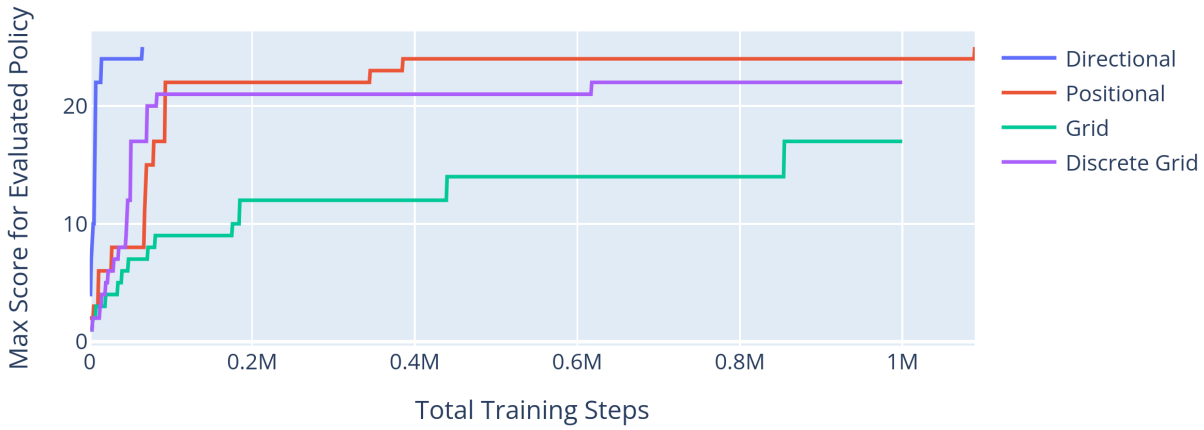

Figure 3: Maximum Scores During Deterministic Policy Evaluation

This result was somewhat surprising at first given the additional information that the grid-based observations included over the simpler ones. However, this was the beginning of forming one of our most important conclusions about this entire experiment, which is that an increase in information does not yield a necessary increase in performance.

The simple truth is that the recurrent neural networks used to train the grid-based observation spaces are far more complex than the simple MLPs used to solve the simpler spaces. For someone with more experience and domain knowledge, having the additional information and a more complex model would probably be advantageous, but we were unable to fully utilize these potential advantages and instead were impeded by the added complexity.

The four runs were given an (approximately) equal number of training steps to keep a level playing field. The exceptions are that the Positional case was given a fraction more than the standard one million steps since it was approaching the winning score of 25, and the Directional case reached the winning score far before this upper limit, so it terminated early.

A tabular summary of the results in the plot above is shown below:

	Maximum Score	**Number of Training Steps**
Directional Space	25	69,100
Positional Space	25	1,089,100
Grid Space	18	1,000,000
Discrete Grid Space	22	1,000,000

4.2. Directional Environment

The directional observation space performed *shockingly* well, especially since we considered it the most rudimentary space of the four. It was able produce a trajectory which obtained the winning score of 25 within 70,000 training steps, which is less than a tenth of the steps that all of the other algorithms were given to train.

Observing the resulting animations of the successful Directional trajectories, we can see the effects of the observation space on the behaviors that arise. The learned policy tends to make frequent, tight turns when adjacent to a terminal state, which is explained by the agent only being able to

observe the cells adjacent to its head. Making tight turns means that there is a smaller chance of the snake cutting itself off by a long straight run it made earlier in the episode, and it was this advantage that allowed the agent to win the game so quickly.

4.3. Positional Environment

The directional observation space also performed better than expected and was able to win the game as well. It took just over one million total steps to produce a winning trajectory, which is over 14 times the number that the directional space took, but it was still impressive to see such a simple action space containing only six integer values perform so well.

In terms of learned behaviors, the trajectories for the Positional policy exhibit less erratic turning than the former case, which makes sense given that it is obtaining information about state that is further away than in the directional case, so it will have more time to react to the information.

4.4. Grid Environment

The grid environment was able to learn a policy that obtained a maximum score of 17. This was an underwhelming result given the high hopes we had for this combination of an information-rich observation space and a neural network that is known to extract positional relationships from grid-like features. We expect that with more time and further study of the inner workings of the RNN/CNN structures, we would be able to improve upon this score.

The Grid space's policy exhibits some strange behaviors and seems to meander aimlessly at some points. It is uncertain whether this is simply a result of the complex parameter space or indicative of poor training on our end.

4.5. Discrete Grid Environment

The discrete grid environment was able to learn a policy that obtained a maximum score of 22. This was a welcome improvement over the normal Grid environment. We were concerned that the increase in the "channel" dimension of the CNN from 1 to 5 would cause issues and increase training times significantly, but neither of these manifested as problems.

We postulate that the increase in performance for the discrete case could be due to the more explicit separation of states due to the one-hot encoded features. In the standard Grid space, the network must learn that a difference in numerical value of one corresponds to an entirely new state, whereas this information is naturally embedded into the one-hot encoded features.

5. Conclusion

5.1. Insights and Recommendations

It is very easy to get caught up in the excitement of the new state-of-the-art due to the rapid and constant advancements in the world of artificial intelligence. There is nothing inherently wrong with this, but when designing experiments, it is important to remain grounded and methodical in our approach. For example, part of the motivation for this project originated from researching other RL applications to *Snake* which utilized fairly simple observation spaces and thinking about how much better a convolutional neural network could perform on the entire state fed as a 2D pseudo-image. We maintain that this is a valid approach – our results confirm it to be so – but in retrospect, it was a mistake to have begun our experimentation on such a complex case. We stress the importance of starting with the simplest feasible solution and building from there. The insight gleaned in simple cases often translates much better to the complex cases than details from complex cases do to simpler ones, and if a valid solution is found, this typically comes with a reduced monetary cost, faster implementation timeline, and a result which is easier to explain, diagnose,

and repair in the event of future failure. Observations which are more information-rich may also require more complex algorithms to tackle, which also requires higher levels of domain expertise, so it is recommended that beginners in the field utilize simpler spaces and incrementally iterate.

When we design simpler observation spaces, they naturally tend to incorporate more of our knowledge about the problem on the front-end. For example, the reason that we designed the positional space the way we did is because we had intuition that if the agent is able to determine how far the terminal states are and where the food is, this would be enough information to achieve its end goal. On the other hand, the grid-based spaces incorporate much more raw information about the problem, but leave it up to the algorithm to learn how to use this data, when much of it may be unnecessary to even use. For example, we feed the entire board state including the walls as observations, when realistically the model should be able to learn that the edges are terminal states even without receiving the walls in the observation. Performance may even improve without including the walls since the algorithm has has only a 5-by-5 grid to process instead of 7-by-7.

Finally, it seems that we tend to underestimate how an algorithm may perform given a limited amount of data in each observation because we impart our own biases on how we would approach the problem. When we play the game, we tend to look at the overall state of the board, attempting to forecast which paths are open in the long run to form an optimal strategy. For this reason, we tend to assume that only having information about the tiles immediately surrounding the head would be too limiting to perform well. Therefore, it is critical to approach experimental design with an awareness of these biases and that we be willing to attempt to first approach the problem with designs that are simple and cost-effective to implement even if they go against our intuition to some degree.

5.2. Future Work

We would like to publish our Gymnasium environment for public use under an open-source license. Ideally, we could submit it to be one of the environments shipped with the Gymnasium package. This will require some significant clean-up of the code before it is in a state to submit back into the main repository, but it would be an exciting way to participate in the RL community.

We believe that one of the most promising areas of future experimentation for this *Snake* environment is multi-agent reinforcement learning (MARL). The rules of the game make it a very interesting test bed for observing different strategic patterns and behaviors in the agents, especially when it comes to competitive vs. collaborative training. Training the agents to compete against each other provides an opportunity to study reward-shaping and how to elicit certain behaviors. We are particularly interested to see whether or not agents can learn to "box each other in" solely by manipulating the reward structure. It also poses an interesting question as to whether competition or collaboration produces the most effective results in such an environment.

Studying these MARL behaviors was one of the initial motivations in formulating this project. Indeed, it was the initial plan that we set forth in our proposal and progress report. However, as we grew further from the ideation phase and nearer to the implementation phase, we decided it would be wise to employ a backup plan for several reasons: the multi-agent problem brings a unique set of challenges, such as the potential loss of stationarity and the need to maintain a training network that is able to communicate simultaneously with different agents. One effect of these challenges is that many of the popular RL libraries (understandably) focus on supporting single-agent training with limited or no support for multi-agent out of the box. TorchRL, the library we plan to use, does provide first-class support for MARL, which is one of the the primary reasons we chose it, but the MARL solutions did require a deeper level of understanding of the library which we simply did not have the time to develop.

Finally, there is more work to be done refining the algorithms for the grid-based observation spaces discussed in this report. We know that there is much more performance that can be drawn out of such an information-rich feature set, but we need to study the inner workings of the RNN and CNN architectures at a deeper level before we can understand where our shortcomings are as of now.

References

[1] W. contributors, "Snake (video game genre) — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/w/index.php?title=Snake_(video_game_genre)&oldid=1261412410

[2] D. Graafsma, "Playing Snake on a Graph." [Online]. Available: http://essay.utwente.nl/102968/

[3] J. J. Garau-Luis, R. K. Goel, E. Crawley, and C. Wu, "Look Ma, No Training! Observation Space Design for Reinforcement Learning." [Online]. Available: https://openreview.net/forum?id=8TyGCAuCGd

[4] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning." [Online]. Available: https://arxiv.org/abs/1312.5602

[5] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning." [Online]. Available: https://arxiv.org/abs/1509.06461

[6] M. Towers *et al.*, "Gymnasium: A Standard Interface for Reinforcement Learning Environments," *arXiv preprint arXiv:2407.17032*, 2024.

[7] A. Bou *et al.*, "TorchRL: A data-driven decision-making library for PyTorch." [Online]. Available: https://arxiv.org/abs/2306.00577